Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems *

Fengguang Song
University of Tennessee
EECS Department
Knoxville, TN, USA
song@eecs.utk.edu

Asim YarKhan
University of Tennessee
EECS Department
Knoxville, TN, USA
yarkhan@eecs.utk.edu

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

ABSTRACT

Multicore systems have increasingly gained importance in both shared-memory and distributed-memory environments. This paper presents a dynamic task scheduling approach to executing dense linear algebra algorithms on multicore systems (either shared- or distributed-memory). We use a task-based library to replace the existing linear algebra subroutines such as PBLAS to transparently provide the same interface and computational function as the ScaLAPACK library. Linear algebra programs are written with the task-based library and executed by a dynamic runtime system. We mainly focus our runtime system design on the metric of performance scalability. We propose an algorithm to solve data dependences without process cooperation in a distributed manner. We have implemented the runtime system and applied it to three linear algebra algorithms: Cholesky factorization, LU factorization, and QR factorization. Our experiments on both shared-memory machines (16-core Intel Tigerton, 32-core IBM Power6) and distributed-memory machines (Cray XT4 using 1024 cores) demonstrate that our runtime system is able to achieve good scalability. Furthermore, we provide analytical analysis to show why the tiled algorithms are scalable and the expected execution time.

1. INTRODUCTION

Multicore systems have increasingly gained importance in both shared-memory and distributed-memory environments [8] [11] [14]. Given a processor with hundreds or even thousands of processing cores, it is critical to increase the degree of thread-level parallelism to utilize all the available cores to improve program performance [1] [4]. The goal is to create as many concurrent tasks as possible to prevent processing cores from becoming idle. The number of synchronization points such as supersteps must be minimized as well since a potentially large number of tasks could be ready to execute but are stalled within the following supersteps.

We strive to design linear algebra software that is going to scale well on both shared-memory and distributed-memory multicore systems. Our approach to developing the scalable software is to put fine-grained computational tasks in a directed acyclic graph (DAG) and schedule them dynamically. To achieve high scalability, we propose a decentralized scheduling scheme for distributed-memory systems. That is, each node runs a private runtime system and communicates with other nodes regarding data dependences only when necessary. The runtime system has no globally shared data structures, no requirement of large storage space for DAGs, and no blocking operations. Furthermore, it respects critical paths and keeps load balance. The runtime system is composed of three types of threads: task-generation thread, computing thread, and communication thread. At any time, only a small portion of the graph is stored in memory. The task-generation thread generates tasks sequentially and stores tasks in a fixed-size *task window* (i.e., building vertices of the graph). The computing thread analyzes the relationship between the tasks in the task window and solves data dependences automatically (i.e., building edges of the graph). The communication thread is responsible for sending and receiving messages.

For easy use, our linear algebra software uses the same interface as `ScaLAPACK`. While offering scalability guarantee, the dynamic DAG scheduling mechanism is transparent to users. Instead of implementing every linear algebra algorithm from scratch, we use a task-based library to generate tasks for the basic linear algebra subroutines such as PBLAS so that a new algorithm is simply a combination of a few task-based subroutines. The execution of a linear algebra program is data-availability driven. It starts from the entry task of the DAG and finishes with the exit task.

We apply the runtime system to a class of tiled linear algebra algorithms: Cholesky factorization, LU factorization, and QR factorization that are introduced in [4]. In the tiled algorithms, each task computes a LAPACK or a Level-3 BLAS subroutine. The tiled algorithms can fully utilize the Level-3 BLAS operations such that the cache hit rate is maximized and data movement is minimized. Our theoretical analysis shows that the tiled LU and QR algorithms have a provably good expected execution time. We conducted experiments on both shared- and distributed-memory machines. Our experimental results demonstrate that the decentralized task scheduling approach is efficient and scalable.

The remainder of the paper is organized as follows. Section 2 introduces linear algebra algorithms and the task-based linear algebra library. Section 3 presents the algorithm to solve data dependences in a distributed manner. Section 4 describes the design and implementation of the runtime system and its space overhead. Section 5 analyzes the expected execution time, the communication and computation ratio, and the degree of parallelism. Section 6 gives the experimental results. Section 7 describes the related work, and finally Section 8 provides our conclusion and future work.

*This material is based upon research supported by the Department of Energy Office of Science under grant Nos. DE-FC02-06ER25761 and DE-ASC05-00OR22725, and by Microsoft Research.

2. TASK-BASED LINEAR ALGEBRA LIBRARY AND PROGRAM

Most LAPACK and ScaLAPACK algorithms are composed of a small number of fundamental operations [2] [5]. The fundamental operations are implemented as Level-2 or Level-3 BLAS routines. For instance, the Cholesky, LU, and QR factorizations all repetitively perform the two operations: panel factorization and trailing submatrix update. The panel factorization transforms the leftmost collection of columns (i.e., column panel) followed by updating the trailing submatrix using the panel factorization result. For instance, Fig. 1 shows an example of block LU factorization. First, the $N \times NB$ panel is factorized by the LU factorization. Next, after some pivoting, we solve the block row U_{12} using L_{11}. Finally, we update the submatrix A_{22} by multiplying the previously computed L_{21} and U_{12}.

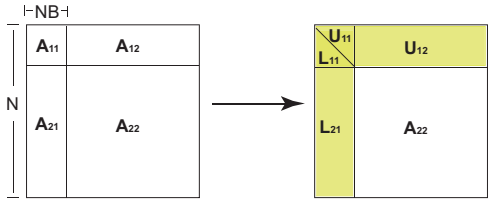

Figure 1: Block LU factorization.

Each fundamental operation can be regarded as a black box. We wish to keep the same interface but execute the program asynchronously in order to eliminate unnecessary barriers between operations (i.e., data-flow driven). To minimize the programming effort, we propose to use a task-based library called Task-based Basic Linear Algebra Subroutines (TBLAS) to replace the fundamental subroutines. The new task-based subroutine does nothing but generate a set of appropriate tasks and store them in a task window (i.e., a fixed size task pool). A runtime system executes the tasks in the task window dynamically.

To understand how it works, Fig. 2 shows the pseudocode that implements the LU factorization with our task-based library. The subroutines of `PDGETF2_T`, `PDTRSM_T`, `PDGEMM_T` simply generate tasks and put them into the task window which is a member of the runtime system data structure `RTS_context`. One can build new linear algebra algorithms directly upon the task-based library.

Programs written with the task-based library will be executed by a single thread called *task-generation thread*. The task-generation thread executes the serial task-based program and creates tasks one by one to keep the original sequential semantics. The number of tasks to be generated are constrained by the size of the task window that is set to be proportional to the number of matrix blocks allocated to a process. Whenever an empty window slot is available, the task-generation thread will start and generate a new task. A finished task will be removed from the task window immediately. Since the task-generation thread does no computation, it takes a very small percentage of the CPU time. The execution of a task-based program is started by the task-generation thread placing the entry task of the DAG into the task window. An idle computing thread picks up and executes the entry task and fires new tasks after finishing it.

```
for (k=0; k < nblks; k++) {
  /* panel factorization */
  PDGETF2_T(nblks-k, 1, A, k, k, &RTS_context);
  /* compute block row of U */
  PDTRSM_T(1, nblks-k, A, k, k, &RTS_context);
  /* trailing submatrix update */
  PDGEMM_T(nblks-k, 1, A, k, k, 1, nblks-k, A, k, k,
           nblks-k, nblks-k, A, k, k, &RTS_context);
}
```

Figure 2: Block LU factorization program written with the task-based subroutines.

3. DISTRIBUTED DEPENDENCE SOLVING

It is not trivial to generate tasks and solve data dependences in a distributed environment without much communication. Processes running on different nodes execute the same program and generate the same set of tasks so that a task may be duplicated by each process. A correct algorithm requires all the processes make a uniform decision regarding which consumer task to fire and how to make sure the consumer task is fired only once. Other complex issues include which process should execute a specific task and how to handle tasks with multiple outputs but belonging to different processes. This section first introduces a centralized algorithm, then describes how to use block data layout and various tasks modes to extend it to a distributed algorithm.

3.1 A Centralized Version

On a shared-memory machine, a single task-generation thread executes the user program sequentially and maintains the serial semantic order between tasks. We use a single linked task list to maintain the task order. If there exists a data dependence, the task list can determine which task precedes another. Figure 3 illustrates how to detect a RAW (read after write), WAR (write after read), or WAW (write after write) dependence based on the task list. A task takes a number of inputs and writes to one or more outputs. Thus, tasks stored in the task list keep information such as the input and output memory locations. Whenever two tasks access the same memory location and one of them is write, the runtime system detects a data dependence and stalls the successor till the predecessor is finished. Since WAR and WAW dependences can be avoided by renaming, we only consider the true dependence (RAW) here.

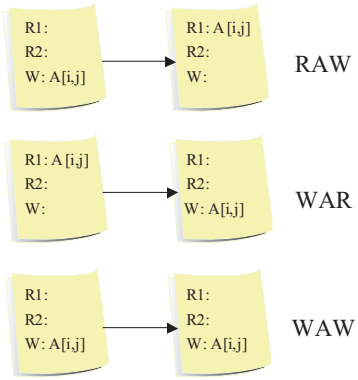

Figure 3: Detecting data dependences based on the task list.

There are two operations to access the task list: *APPEND* and *FIRE*. The task-generation thread generates a new task t_j and invokes the APPEND operation to put task t_j to the end of the task list. Before the appending, APPEND scans the task list from the list head to check if there exists a task t_i such that t_i writes to datum x and t_j reads datum x (i.e., a data dependence). If none of the previous tasks writes to t_j's kth input, we set the status of t_j's kth input as "ready". When all of t_j's inputs are ready, task t_j becomes a ready task.

After a task completes and modifies datum y, the FIRE operation starts to search for the tasks that want to read the datum y. Instead of from the head of the list, the runtime system scans the task list from the position of the completed task to the end of the list to find which tasks are waiting for y. The scanning process will exit when confronting the first task that writes to y. We denote the set of scanned tasks that are linked between the completed task and the exit point as S. If a task is in S and one of its inputs is datum y, the FIRE operation marks that input as "ready". If the completed task has more than one output, the FIRE operation will do the same task list scanning but check dependences for all the outputs simultaneously. Since we track data dependences for data blocks and use a fixed size task window, the space overhead is not expensive. Section 4.3 discusses the space overhead of this method.

3.2 Block Data Layout and Task Assignment

Block data layout is a technique used to improve memory hierarchy performance [12]. In the block data layout, a matrix of size N is divided into submatrices (or blocks) of size NB × NB. Data elements within a block are stored contiguously in memory. On a distributed memory system, we use the 2D cyclic distribution method to map matrix blocks to different processes. The *process grid* is used to map a 1D array of P processes to a 2D rectangular grid (or mesh). We assume a process grid has P_r rows and P_c columns, where $P_r \times P_c = P$. Let $A[I, J]$ be a matrix block that is located at the I-th row and J-th column of matrix A, then $A[I, J]$ will be mapped to process $[I \ mod \ P_r, J \ mod \ P_c]$. Note that we always map a block as an indivisible unit. We also bind a task to its output block such that the computation is centered around data to minimize data movement and maximize data locality. If the output of a task t is $A[I, J]$, then we assign task t to process $[I \ mod \ P_r, J \ mod \ P_c]$. Since blocks are allocated to processes statically, the assignment of tasks to processes is static too. The task scheduling within a process is dynamic.

In addition to reducing the data movement cost, the 2D cyclic distribution of blocks and tasks has a few more advantages. It has been proven to have the following properties [6] [9]: (1) Communication volume is within a constant factor of the optimal. (2) The maximal load imbalance is $N^2(P_r + P_c - 2)/(2P)$, which is small compared to the runtime $\Theta(N^3/P)$. (3) The proportional load imbalance stays constant while increasing the number of processes. (4) Matrix size N is capable of growing with $\sqrt{P}$ to maintain scalability when we increase the number P of processes considering the fact that the matrix memory requirement grows with N^2. Furthermore, it allows us to design a compact efficient runtime system to avoid complex cases such as distributed work stealing and dynamic tracking of the ownership of blocks. We believe the property of the bounded load imbalance is able to provide a balanced workload on every process. The ScaLAPACK library ([2]) and our experiments in Sect. 6 demonstrate that using the 2D cyclic distribution can achieve good load balance and high performance.

3.3 Various Task Modes

It is sufficient to create a single instance for a task on shared-memory machines. The single instance contains all the necessary information for the runtime system to analyze data dependences and execute the task. A task contains the following information:

- Task-related information such as task id, function type, and priority.
- Input: which blocks are the inputs and a ready status for each input. A block is denoted by a 3-tuple of ⟨matrix, row index, column index⟩.
- Output: which block is the output. If a task has more than one output, we distinguish them as *minor* outputs.

On distributed-memory machines, a task will correspond to a number of task instances since each task will be created and inspected by all the processes. We assume a task has a constant number of inputs and outputs and propose a novel approach to generating tasks. The objective is to make all the processes reach the same conclusion without any cooperative communication.

Suppose a task has c_1 inputs and c_2 outputs, then we create a number $c_1 + c_2$ of task instances and distribute them to different processes. Each task instance plays a role of "representative" for the task's corresponding input or output. The location of the task instance follows the location of the represented input or output. As defined in Table 1, the major output of a task corresponds to an *owner* task instance and it is the process who stores the owner task instance to execute the task. Each input of a task corresponds to an *input shadow* task instance (either *local* or *remote* depending on whether the input and the major output are assigned to the same process). If a task has more than one output, then for each minor output, we either create a *local minor-output-shadow* task instance or a pair of *source/sink minor-output-shadow* task instances. When the minor output and the major output are stored in the same process, the minor output leads to a *local* minor-output-shadow task instance. Otherwise, one *source* and one *sink* minor-output-shadow task instances are generated and stored in two processes, respectively. Note that the location of an input or output is well defined based on the 2D cyclic distribution method.

3.4 The Distributed Algorithm

We partition the task list described in the previous centralized version into multiple lists across different processes. Each process maintains a private task list. To reduce the time to traverse the task list, we further divide a process's private task list into a number of *block access lists* so that each block $A[I, J]$ is associated with a separate task list. This way, we can perform the APPEND and FIRE operations quickly on shorter lists. Section 4.3 discusses the memory requirement to store block access lists.

The distributed algorithm predefines an arbitrator for each block to decide the data dependence involving the specific block. At any time, only the block's arbitrator can make

Table 1: A variety of task modes.

Task mode	Definition
Owner	An owner task instance is stored by the process which owns the task's major output. The owner task instance keeps the complete information of the task.
Local input shadow	A local input shadow task instance is stored by the process which owns the specific input. The input block and the task's major output block must belong to the same process. The local input shadow task instance keeps partial information regarding which specific input to read and a pointer to the owner task instance.
Remote input shadow	A remote input shadow task instance is stored by the process which owns the specific input. The input and the task's major output must belong to different processes. The remote input shadow instance keeps partial information about which input to read and what is the output.
Local minor output shadow	A local minor-output shadow task instance is stored by the process which owns the minor output. The minor-output and the task's major output must belong to the same process. The task's owner task instance keeps a pointer pointing to the local minor-output shadow.
Source minor output shadow	If the minor-output of a task belongs to a process different from the task's major output, a source minor-output shadow task instance is generated and stored by the owner task's process. The source minor-output shadow instance keeps partial information regarding what is the minor-output block. The task's owner instance keeps a pointer to the source minor-output shadow.
Sink minor output shadow	A sink minor-output shadow task instance is stored by the process to which the minor output is assigned. The availability of the sink minor-output shadow is notified by the availability of the source minor-output shadow.

the decision. We let the process that owns the block be the arbitrator and determine data dependences for its owned set of blocks. We extend the previous centralized algorithm to the distributed algorithm using the following rules:

- Both blocks and tasks are allocated to specific processes by 2D cyclic distribution.
- Every process has a task-generation thread and generates tasks independently.
- Every process only stores and keeps track of matrix blocks assigned to itself.
- Every process stores "relevant" tasks only. That is, suppose a task t takes block $A[I, J]$ as an input or output, $A[I, J] \in$ process P_i implies that an instance of task t will be created and stored by P_i. A particular task mode will be assigned to the task instance based on Table 1.

We now use a simple example to show how the distributed algorithm works (Fig. 4). Suppose a matrix of size 3 blocks by 3 blocks is distributed to a 2×2 process grid by 2D cyclic distribution, then each process is allocated with a set of blocks (i.e., shaded blocks). Let the processes P1, P2, P3, P4 execute a sequential program and generate a set of tasks: t_1, t_2, and t_3. We assume task t_1 reads and writes block 1, task t_2 reads block 1 and writes block 4, and task t_3 reads block 1 and writes block 7. Figure 4 illustrates what task instances for t_1, t_2, t_3 are generated and where they are stored. Since every task has one output, only three task modes are used in the example. Based on the status of the task lists on P1 and P3, it is easy to find that t_2 and t_3 can be started simultaneously when task t_1 is finished.

THEOREM 1. *The distributed algorithm guarantees that a task will eventually get all of its inputs and become ready.*

PROOF SKETCH. Suppose a task T $\in P_0$ has k inputs which are allocated on k different processes $\{P_1, \ldots, P_k\}$. Let task T's ith input be generated by P_i at time t_i and be received by P_0 at time t_i'. Also suppose T is generated by P_0 at time t_0. There could be an arbitrary order in the set $\{t_0, t_1', t_2', \ldots, t_k'\}$. To prove task T can eventually get all of its inputs and become ready, we need to show the distributed algorithm handles the following three cases correctly:
Case 1: $t_0 < \{t_1', t_2', \ldots, t_k'\}$. The received input simply updates the corresponding ready status for task T.
Case 2: $\{t_1', t_2', \ldots, t_k'\} < t_0$. The runtime system creates a temporary task until P_0 replaces it by T at time t_0.
Case 3: $\{t_1', \ldots, t_s'\} < t_0 < \{t_{s+1}', \ldots, t_k'\}$. It is equivalent to the mixed case of 1 and 2. □

THEOREM 2. *The distributed algorithm is deadlock free for any task window of size $W \geq 1$.*

PROOF. Each process P_i has a task window Q_i. Suppose a deadlock occurs between m processes $\{P_{i_1}, P_{i_2}, \ldots, P_{i_m}\}$

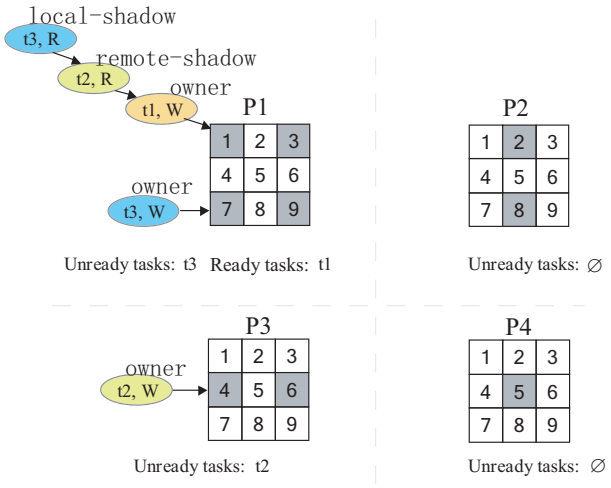

Figure 4: Snapshot of the distributed algorithm after the processes generated tasks t_1, t_2, t_3.

that form a waiting cycle such that P_{i_k} waits for $P_{i_{k+1}}$. If the first task in the task window is a non-owner task, it will be executed and removed immediately. So when the deadlock happens, the first task $\in Q_i$ must be an owner task. Let t_{i_d} be the first task in Q_{i_d}, where $d \in [1, m]$. Suppose $t_{i_1} \in Q_{i_1}$ is unable to execute because t_{i_1} is waiting for one of its parent task $\wp(t_{i_1}) \in Q_{i_2}$ to finish. Task $\wp(t_{i_1})$ must be either t_{i_2} itself or behind t_{i_2} in Q_{i_2}. Thus, $t_{i_2} \leq \wp(t_{i_1}) < t_{i_1}$. By following the deadlock cycle of $\{P_{i_1}, P_{i_2}, \ldots, P_{i_m}, P_{i_1}\}$, we can show that $t_{i_1} < t_{i_m} < t_{i_{m-1}} < \ldots < t_{i_2} < t_{i_1}$. It contradicts the fact that each task window stores the tasks in the program's sequential order. □

4. RUNTIME SYSTEM DESIGN

On a distributed-memory system, every compute node runs an instance of the runtime system. The runtime system has two task pools: a task window and a ready task pool. The task window stores all the generated but not finished tasks. The implementation of the task window actually uses the *block access lists* indexed by block locations $[I, J]$. The maximal number of tasks to be generated is constrained by the task window size. As introduced in Sect. 3.4, a process's runtime system only stores the blocks and tasks that are assigned to the process based on the 2D cyclic distribution. The ready task pool is much simpler than the task window. It stores a pointer pointing to the corresponding ready task in the task window (Fig. 5). Each task has a priority. Hints regarding critical paths (e.g., panel tasks in the factorizations) are provided by the task-based library writers and tasks on the critical path are assigned a high priority.

4.1 Thread Types

There are three types of threads in the runtime system: task-generation thread, computing thread, and communication thread. If a node has n processing cores, we launch one task-generation thread, one communication thread, and $n-1$ computing threads. We let the $n-1$ computing threads occupy $n-1$ cores. As shown in Fig. 5, the task-generation thread executes a sequential program and generates tasks to fill in the task window by invoking the APPEND operation. The task-generation thread uses a counting semaphore to start or stop itself depending on whether the task window is full or not. Whenever a computing thread becomes idle, it picks up a ready task from the ready task pool and computes it. After finishing the task, the computing thread will perform the FIRE operation to solve dependences and find the finished task's corresponding children.

The communication thread is responsible for sending and receiving messages by posting `MPI_ISend` and `MPI_IRecv` operations. The interaction between the computing threads and the communication thread is through the message `inbox` and `outbox`. If a computing thread wants to send a block to some computing threads running on different nodes, it puts a message in the `outbox` and the communication thread will send it out. Whenever receiving a message, the communication thread places the message in the `inbox` which will be read by one of the computing threads.

The communication thread and the task-generation thread take the last core. The reason why we do not launch multiple communication threads is because the thread support level of MPI_THREAD_MULTIPLE at the moment is not portable on all systems and mixing computing thread and communication thread together on the same core interferes

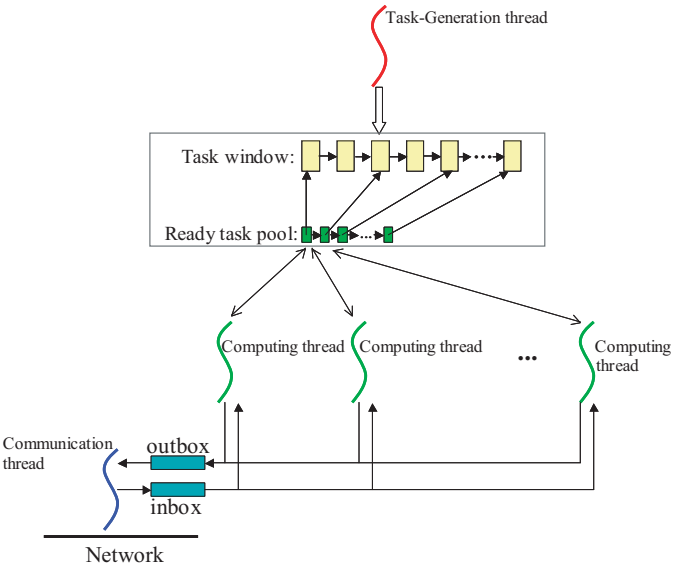

Figure 5: Architecture of the runtime system.

with the maximized cache hit rate of the computing thread. When there are many cores on each node (e.g., 16 or more), it is reasonable to dedicate one of the many cores to process communications.

4.2 Memory Deallocation

We use an indirect data structure to store matrices. Given a matrix A of size N and block size of NB, the indirect data structure consists of $(\frac{N}{NB})^2$ pointers pointing to a number $(\frac{N}{NB})^2$ of NB × NB blocks for matrix A. There are two matrix types in our runtime system: user-defined input/output matrix and intermediate-result matrix. The intermediate-result matrices are allocated and deallocated on demand by the runtime system. The first task that writes to the intermediate block will allocate memory for the block. We assume there is always available memory to allocate. If memory allocation fails, the runtime system returns an error and aborts.

It is more difficult to deallocate blocks because the runtime system cannot decide whether a block will be used or not in the future. Similar to ANSI C programs calling `free()` to release memory, we provide programmers with a special routine `Release_Block()` to free a block. `Release_Block()` actually doesn't release any memory, but sets up a marker in the task window. While generating tasks, the task-generation thread keeps track of the expected number of visits for each block. Meanwhile the computing thread records the actual number of visits for each block. The runtime system will free a block if and only if the following three conditions are satisfied: i) The block is currently stored by the process. ii) The actual number of visits is equal to the expected number of visits to the block. iii) `Release_Block` has been called to free the block. In our runtime system, each block maintains three data members for the memory deallocation: `num_expected_visits`, `num_actual_visits`, and `is_released`. This memory deallocation method bridges the gap between the on-the-surface deterministic programs and the internal nondeterministic execution by the dynamic runtime system based on data availability.

4.3 Space Overhead

This section analyzes the memory requirement of the runtime system for keeping track of data dependences. For every input and output of a task, there is a task instance generated and added to the task list. The owner task instance stores the complete information of the task. Suppose the owner task has k arguments, then it uses $3k \times 4$ bytes since each argument is represented by 3 integers. A non-owner task instance stores information of task id, block location, and a flag of input or output (i.e., 17 bytes). Therefore, every task corresponds to $12k + 17(k-1)$ bytes. If the task window size is W, the runtime system uses $W(29k - 17)$ bytes to keep the data dependence related information.

Although a small task window size saves memory space, a larger window size can explore more tasks in a longer distance and identify more parallelism. In our implementation, we choose the task window size to be equal to the number of blocks assigned locally to each process. Note a matrix is distributed across processes by 2D cyclic distribution. Suppose a compute node has a memory of capacity M bytes (e.g., 4 GBytes). Let NB be the block size, then the local matrix has a maximum dimension of $\sqrt{M/8}/\text{NB}$ blocks and the task window size W is $M/8/\text{NB}^2$. The ratio of the space to store the W tasks over the M-bytes is thus $(29k - 17)/8/\text{NB}^2$. To avoid too fine-grained tasks, we expect NB to be at least 32. Suppose a sufficiently complex task has a number k=16 of arguments, the overhead is just about 5.5% for NB=32. The overhead will become much smaller when NB is bigger and k is smaller. For instance, if k is still 16 but NB=100, the space overhead is equal to 0.56%.

5. ANALYSIS OF SCALABILITY

We use the tiled algorithms presented in [4] to implement the Cholesky, LU, and QR factorizations. Although the experimental results in the next section will demonstrate the scalability of these algorithms, we also wish to prove that the algorithms are scalable in theory and analyze what is the expected execution time and to what extent the algorithms could continue to scale.

The Cholesky factorization algorithm has a very high degree of parallelism where each finished task in the panel can fire a number nb of tasks in the trailing submatrix, where nb is the matrix dimension in blocks. The LU and QR factorizations are updating-based algorithms whose data dependence graphs are much denser than that of the Cholesky factorization. This section skips the simpler Cholesky factorization algorithm and analyzes the performance of the updating-based LU and QR factorization algorithms.

In both LU and QR factorizations, the trailing matrix update occupies the most of the computation. Figure 6 shows an example of updating a matrix of size 4 blocks × 4 blocks on a 2×2 process grid. Each task in the ith row is dependent on the task in the (i-1)th row and all the tasks on the same row are totally independent. For details of the algorithms, please refer to [4].

5.1 Expected Execution Time

THEOREM 3. *Suppose each task takes the same amount of time to compute and tasks on the ith row are dependent on tasks on the $i-1$th row. The tasks located on the same row have no data dependences. If $nb \gg P$, the expected execution time T is as follows:*

$$T = \frac{nb^2(t_{comp} + \frac{t_{comm}}{k})}{P_r P_c} + 2k(P_r - 1)t_{comp}, \text{ where}$$

nb is the matrix dimension in blocks, t_{comp} is the computation time of a task, t_{comm} is the communication time for a tile, k denotes a virtual tile of $k \times 1$ tiles, and $P = P_r \times P_c$ is the process grid.

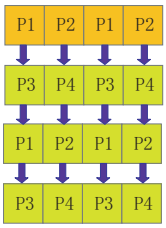

Figure 6: Trailing submatrix update in the tiled LU and QR algorithms.

PROOF. The tasks in the trailing matrix update are essentially executed along a pipeline. Each process occupies a set of stages in the pipeline (Fig. 7). Consider an arbitrary process P_i. At time $t = 0$, P_i has $nb/P_c \times k$ tasks. The next time P_i appears in the pipeline is at $t + P_r \times (t_{comp} \times k + t_{comm})$ when P_i will get another $nb/P_c \times k$ tasks. Since $nb \gg P$, P_i will get new tasks continuously and never become idle. The expected execution time

$$T = T_{computation} + T_{communication} + T_{pipestart} + T_{pipefinish}.$$

$$\text{We know } T_{computation} = (\frac{nb^2}{P_r P_c})t_{comp},$$

$$T_{communication} = (\frac{nb^2}{P_r P_c \times k})t_{comm},$$

$$T_{pipestart} = T_{pipefinish} = k(P_r - 1)t_{comp},$$

thus

$$T = \frac{nb^2(t_{comp} + \frac{t_{comm}}{k})}{P_r P_c} + 2k(P_r - 1)t_{comp}.$$

□

Figure 7 shows an example of the asynchronous pipeline execution on a 4×4 process grid. In the example, P0 gets new tasks at time $t = 5$ after P4, P8, P12 finish their tasks in sequence. Each rectangle in the figure represents a virtual tile that consists of 4×1 tiles.

COROLLARY 1. *Let T_1 be the total computation time (i.e., $nb^2 \times t_{comp}$). If $nb \gg P$, then the expected execution time*

$$T = \frac{T_1}{P}(1 + \frac{t_{comm}}{kt_{comp}} + \frac{2k(P_r - 1)P}{nb^2}) \cong \frac{T_1}{P}(1 + \frac{t_{comm}}{kt_{comp}}).$$

5.2 Communication and Computation Ratio

In the tiled linear algebra algorithms, each task computes a Level-3 BLAS operation. A Level-3 BLAS operation has a time complexity of $O(\text{NB}^3)$. This section analyzes the ratio for the frequently used `DGEMM` whose time complexity is

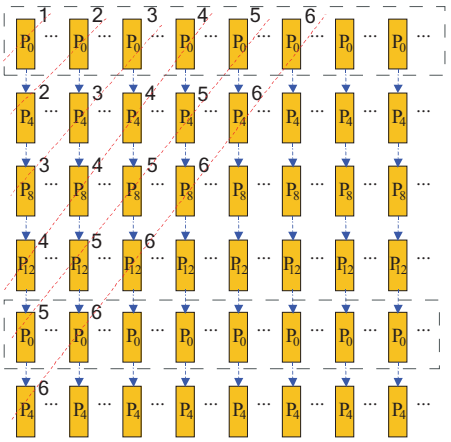

Figure 7: Pipeline execution by a 4×4 process grid.

2NB^3. Other Level-3 operations may have different complexities such as $\frac{1}{3}\text{NB}^3$, $\frac{2}{3}\text{NB}^3$, and so on. The formula of $\frac{t_{comm}}{kt_{comp}}$ for `DGEMM` can be expressed as follows:

$$\frac{t_{comm}}{kt_{comp}} = \frac{8 \times \text{NB}^2/\text{bw}}{2\text{NB}^3 \times k/\text{flops}}, \text{ where}$$

bw denotes the injection network bandwidth in GB/s, and *flops* denotes the maximum `DGEMM` performance per core in GFLOPS.

We now look at the value of $\frac{t_{comm}}{kt_{comp}}$ for two real machines. The first machine is a cluster machine connected by a Myrinet network. Each core on the cluster has a maximum `DGEMM` performance of 10 GFLOPS and an injection network bandwidth of 1.2 GB/s. Based on the above formula, to keep the ratio $\frac{t_{comm}}{kt_{comp}} < 10\%$ on the cluster machine, we should set NB ≥ 320 for k=1, NB ≥ 160 for k=2, and NB ≥ 80 for k=4, accordingly. The second machine is a Cray XT4 machine, where flops=8 GFLOPS and bw=4 GB/s. On the Cray XT4 machine, setting NB = 80 and k=1 is sufficient to attain a small ratio of 10%. Therefore, different systems may require different NB sizes to achieve good program performance.

5.3 Degree of Parallelism

This section discusses the condition for which the tiled updating algorithms can achieve good scalability if we double the number of cores constantly. As described in Theorem 3, each process in the pipeline continuously receives new tasks. Suppose a matrix of dimension nb blocks is distributed across a $P = P_r \times P_c$ process grid, and each process has C threads running on C cores. Suppose the algorithm executes a number nb of iterations. In the ith iteration, process p appears in a number $(nb-i)/P_r$ of rows. Also process p appears $(nb-i)/P_c$ times on each row. The tasks within each row are totally independent and can be executed in parallel, but the tasks between rows must be executed in sequence. We let *TaskGain* represent the number of ready tasks in process p. Between every two appearances of process p on two rows, there are a maximal number $\frac{(nb-i)d}{P_c}$ of tasks entering process p assuming a lookahead depth of d. But the C threads of process p have also computed (or consumed) $P_r \times C$ tasks since the distance between the two rows is P_r tasks. Therefore, in the ith iteration,

$$TaskGain = \frac{(nb-i)d}{P_c} - P_r \times C.$$

Because the algorithm has nb iterations and $i = 0 \ldots nb-1$, the overall task gain of process p can be expressed as:

$$TotalTaskGain \cong \frac{nb^2}{6P}(2(1+d)nb - 3 \times P \times C).$$

Note that as long as $(1+d)nb \geq 1.5 \times TotalNumberCores$, every core will keep receiving new tasks constantly and not become idle. We can also conclude that the lookahead technique is able to improve the degree of parallelism and is necessary for the asynchronous algorithms.

6. EXPERIMENTAL RESULTS

To evaluate the effectiveness of our dynamic scheduling approach and runtime system design, we conducted experiments on both shared-memory and distributed-memory multicore systems.

6.1 Shared-Memory System

We applied our runtime system to the Cholesky factorization and the QR factorization on two different multicore SMP machines: a 16-core Intel Tigerton machine and a 32-core IBM Power6 machine.

Figures 8 and 9 show our measurements on the Intel Tigerton machine with 16 2.4-GHz cores (4 sockets, 4 cores each socket) and 32GB memory. We compiled our dynamic scheduling programs (called TBLAS) with Intel Fortran and C/C++ 11.0 compilers at optimization level -O3. We compared TBLAS to three libraries: LAPACK, Intel MKL 10.1, and PLASMA 1.0. PLASMA is a parallel linear algebra library developed at University of Tennessee for shared-memory multicore architectures [15]. To get a feeling of the best performance upper bound, we also list the performance of the serial DGEMM multiplied by sixteen cores (labeled as $16 \times$ dgemm-seq). As for Cholesky factorization (Fig. 8), TBLAS is slightly better than Intel MKL but not as good as PLASMA. When the matrix size is large, TBLAS has the same performance as PLASMA. LAPACK is not able to provide a good performance on the multicore machine. As seen in Fig. 9 which shows the performance of QR factorization, TBLAS is better than PLASMA when matrix size becomes large. Both TBLAS and PLASMA are much faster than Intel MKL 10.1.

The second multicore SMP system is an IBM Power6 machine with 32 4.7-GHz cores (4 Multi-Chip Modules (MCM), 4 dual-core chips on each MCM). We compiled the TBLAS programs with IBM `xlf 11.1` and `xlc 9.0` compilers. Figures 10 and 11 compare the performance of TBLAS to three libraries: LAPACK, IBM ESSL 4.3, and PLASMA 1.0. The performance of DGEMM is also displayed to show the best upper bound. For Cholesky factorization (Fig. 10), TBLAS is significantly better than ESSL and LAPACK, but slower than PLASMA. As for QR factorization (Fig. 11), TBLAS is better than ESSL when matrix size is greater than 6000 and the same as PLASMA when matrix size is greater than 10,000 by using 32 processing cores.

PLASMA 1.0 uses hand-optimized static schedules to implement each linear algebra algorithm, hence it incurs less overhead and solves relatively small matrices more efficiently than the TBLAS runtime system. Although the performance of TBLAS is not as good as that of PLASMA for

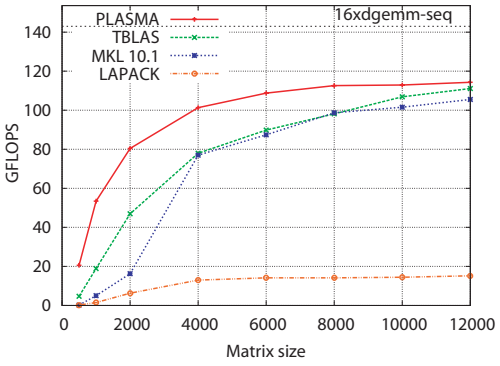

Figure 8: Cholesky factorization on Intel Tigerton.

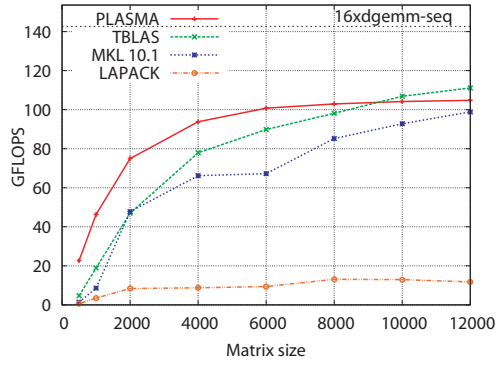

Figure 9: QR factorization on Intel Tigerton.

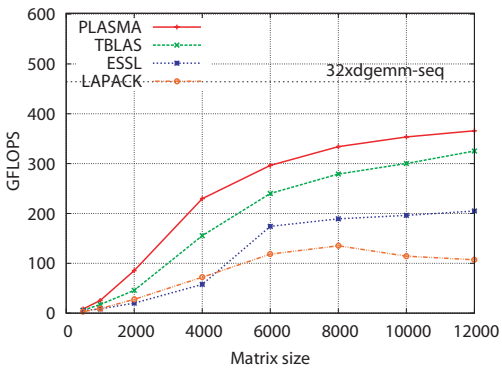

Figure 10: Cholesky factorization on IBM Power6.

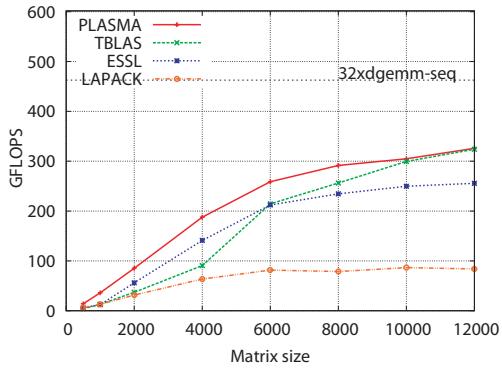

Figure 11: QR factorization on IBM Power6.

smaller matrices, TBLAS provides a comparable performance when matrix size is large. Also it should take much less time and effort to redesign the other algorithms in LAPACK or ScaLAPACK by using the TBLAS runtime system than by the hand-optimized static scheduling method. Moreover, TBLAS is intended to provide scalable performance on distributed-memory multicore systems.

6.2 Distributed-Memory System

We measured the performance of our runtime system on the Cray XT4 Jaguar machine from ORNL. The machine consists of nearly 8000 compute nodes each of which has a quad-core 2.3-GHz AMD Opteron processor and 8GB memory. Cray XT4 adopts a 3D torus topology and is connected by a SeaStar2 network. On this machine, the peak performance per core is 9.2 GFLOPS and the maximum DGEMM performance per core is 7.6 GFLOPS. We will look at the weak scalability performance of our runtime system. That is, when we double the number of nodes, we also increase the matrix size N accordingly. Since the matrix memory requirement grows with N^2 but the physical memory size grows linearly with the number of nodes, we increase the matrix size by $\sqrt{2}$ when we double the number of nodes. The first matrix size for our single node experiment is $20,000$. To minimize message delays in our runtime system, we dedicate one core on each node to do nothing but MPI communications. Therefore, we just used 3 out of 4 cores (25% less) on each node to do real computations. However, for the per-core performance, we divide TBLAS's overall performance by $4 \times$ NumberNodes, instead of $3 \times$ NumberNodes. We believe a node with more than 16 cores can achieve a much better performance (i.e., 1/16=6.25% less).

We compare TBLAS to the ScaLAPACK library provided by Cray `XT-LIBSCI` 10.3.2. We implemented three types of matrix factorizations: Cholesky, LU (with and without pivoting), and QR. For each factorization, TBLAS uses two configurations. One is a single shared-memory node which uses 4 cores for computation. The other is a distributed-memory many-node configuration where each node uses 3 cores for computation and 1 core for communication. Therefore, in Figs. 13, 15, and 17, TBLAS has two separate lines for the two configurations, respectively.

Figures 12 and 13 demonstrate the overall and per-core performance of the Cholesky factorization. The per-core performance is the overall performance divided by the number listed on the x-axis. Although TBLAS uses 25% less cores than ScaLAPACK for computation, it is close to ScaLAPACK (Fig. 12). In Fig. 13, the `4-comp-core` performance decreases from 6.8 GFLOPS to the `3-comp-core` 5.2 GFLOPS, which is 23% less than the `4-comp-core` performance. For more than one compute node, TBLAS is scalable from 8 cores to 1024 cores.

Figures 14 and 15 present the results for the LU factorization. We list the LU factorization both with and without pivoting. Since the LU without pivoting uses an algorithm close to Cholesky factorization, its performance is as good as that of Cholesky factorization. As shown in Fig. 15, TBLAS LU with pivoting again scales well from 8 to 1024 cores. Moving from `4-comp-core` (i.e., 5.1 GFLOPS) to `3-comp-core` (i.e., 4 GFLOPS), the TBLAS performance is

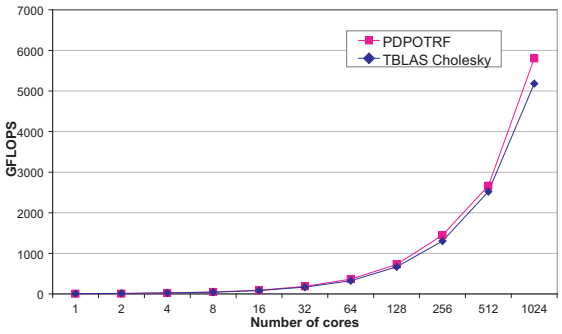

Figure 12: Overall performance of Cholesky factorization on Cray XT4.

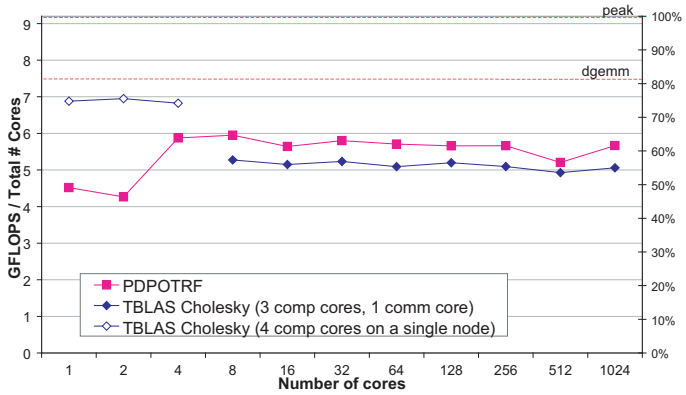

Figure 13: Weak scalability of Cholesky factorization on Cray XT4.

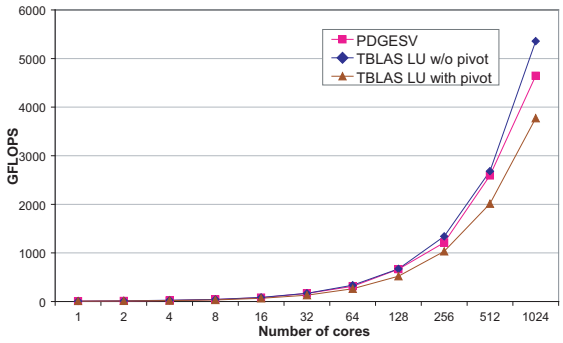

Figure 14: Overall performance of LU factorization on Cray XT4.

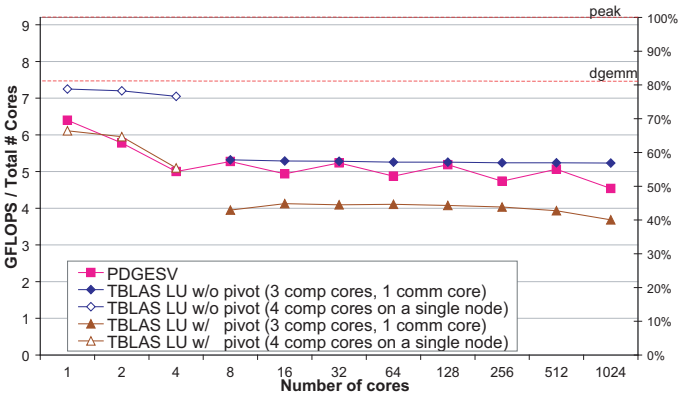

Figure 15: Weak scalability of LU factorization on Cray XT4.

decreased by 21%. The performance of TBLAS LU with pivoting on a single node drops greatly from two cores to four cores. We are currently working on how to tune certain parameters in the computational kernels to improve its performance.

Figures. 16 and 17 show the performance of the QR factorization. As seen in Fig. 17, the difference between TBLAS `4-comp-core` and `3-comp-core` is equal to 25% (5.7 GFLOPS vs 4.25 GFLOPS), which is just equal to the 25% less compute cores that TBLAS uses than ScaLAPACK.

In summary, our experiment scales from 1 core to 1024 cores on a quad-core distributed-memory machine. At first glance it might appear that TBLAS is inferior to ScaLAPACK. But if a compute node has more than 16 cores, we expect that the TBLAS Cholesky factorization will provide a much better performance than ScaLAPACK since the present 25% unutilized compute cores (1 out of 4 cores) will become less than 6.25% (Fig. 13). Similarly, we expect the performance of the TBLAS LU and QR factorizations to be comparable to ScaLAPACK's performance since using more than 16 cores per node will make TBLAS LU and QR factorizations reach the single node performance (Figs. 15 and 17).

7. RELATED WORK

Most of the work that uses dynamic task scheduling has focused on shared-memory systems. Cilk is a multithreaded language that generalizes the semantics of C and uses a provably good "work-stealing" algorithm to schedule tasks. The Cilk programming model is mainly used to solve recursive problems [3] [7]. Buttari et al. designed and implemented a set of linear algebra algorithms for multicore machines [4] [10]. Their algorithms use the block data layout and schedule fine-grained tasks dynamically. Recently they extended their work and developed a library called PLASMA for a richer set of linear algebra algorithms [15]. The initial release of PLASMA 1.0 expresses each algorithm's data dependence relationship manually and schedules tasks statically.

SMP Superscalar is also a parallel programming environment for multicore architectures. SMP Superscalar compiles a sequential C program and links it with the runtime system, and executes the program in parallel. Similar to TBLAS, it is able to analyze data dependence at runtime [13]. Instead of using compiler technology, TBLAS replaces the basic linear algebra routines by a task-based library to run programs in parallel automatically. Most importantly, our work is focused on designing scalable software for distributed-memory systems. In addition, the same TBLAS program is able to work on both shared-memory and distributed-memory multicore systems in an efficient and scalable manner.

8. CONCLUSIONS AND FUTURE WORK

We have designed a runtime system to schedule tasks dynamically on both shared- and distributed-memory multicore systems. Linear algebra programs are written with

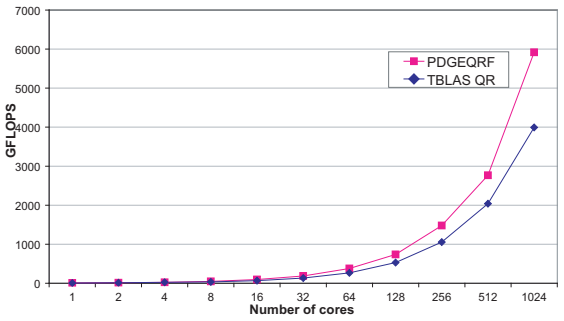

Figure 16: Overall performance of QR factorization on Cray XT4.

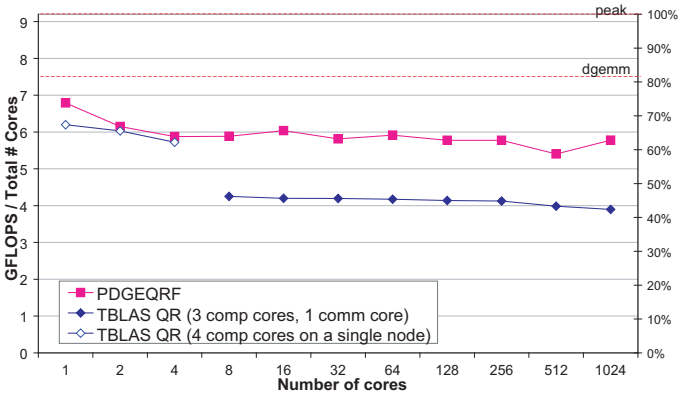

Figure 17: Weak scalability of QR factorization on Cray XT4.

a task-based library and can be executed by the runtime system automatically. We mainly focus our runtime system design on the scalability metric. The paper has the following contributions: (i) proposes a task-based library to support dynamic scheduling of linear algebra algorithms automatically; (ii) presents a distributed algorithm to resolve data dependences without process communication; (iii) demonstrates the scalability and practicality of the dynamic scheduling approach on both shared-memory and distributed-memory multicore systems; (iv) proves that the class of tiled linear algebra algorithms have sufficient degree of parallelism and can offer scalability guarantees.

The current runtime system prototype does not optimize memory affinity between tasks because when the block size NB is large (e.g., ≥ 80), an optimized subroutine such as Level-3 BLAS can maximize the cache hit rate and achieve high performance. For small NBs, we plan to add new features to the runtime system to improve the memory affinity. Another research along this line is to study the dynamic scheduling approach on NUMA architectures.

9. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 5, Washington, DC, USA, 1996. IEEE Computer Society.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[4] 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, 2009.

[5] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Sci. Program.*, 5(3):173–184, 1996.

[6] J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable dense linear algebra libraries. *Scalable High Performance Computing Conference. SHPCC-92. Proceedings.*, pages 372–379, Apr 1992.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[8] J. Gray. The Roadrunner supercomputer: A petaflop's no problem. *Linux J.*, 2008(175):1, 2008.

[9] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Comput.*, 15(5):1201–1226, 1994.

[10] 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(9):1175–1186, 2008.

[11] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM Power6 microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662, 2007.

[12] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, 2003.

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

[14] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, and M. Abrash. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[15] University of Tennessee. PLASMA. `http://icl.cs.utk.edu/plasma`, 2009.