Scaling Up Matrix Computations on Shared-Memory Manycore Systems with 1000 CPU Cores [*]

Fengguang Song
Department of Computer Science
Indiana University-Purdue University
Indianapolis
fgsong@cs.iupui.edu

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

ABSTRACT

While the growing number of cores per chip allows researchers to solve larger scientific and engineering problems, the parallel efficiency of the deployed parallel software starts to decrease. This unscalability problem happens to both vendor-provided and open-source software and wastes CPU cycles and energy. By expecting CPUs with hundreds of cores to be imminent, we have designed a new framework to perform matrix computations for massively many cores. Our performance analysis on manycore systems shows that the unscalability bottleneck is related to Non-Uniform Memory Access (NUMA): memory bus contention and remote memory access latency. To overcome the bottleneck, we have designed NUMA-aware tile algorithms with the help of a dynamic scheduling runtime system to minimize NUMA memory accesses. The main idea is to identify the data that is, either read a number of times or written once by a thread resident on a remote NUMA node, then utilize the runtime system to conduct data caching and movement between different NUMA nodes. Based on the experiments with QR factorizations, we demonstrate that our framework is able to achieve great scalability on a 48-core AMD Opteron system (e.g., parallel efficiency drops only 3% from one core to 48 cores). We also deploy our framework to an extreme-scale shared-memory SGI machine which has 1024 CPU cores and runs a single Linux operating system image. Our framework continues to scale well, and can outperform the vendor-optimized Intel MKL library by up to 750%.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*

[*]This research was supported by an allocation of advanced computing resources at the National Institute for Computational Sciences and the National Science Foundation.

1. INTRODUCTION

The number of cores per chip keeps increasing due to the requirements to increase performance, to reduce energy consumption and heat dissipation, and to lower operating cost. Today, more and more multicore compute nodes are deployed in high performance computer systems. However, the biggest bottleneck for the high performance systems, is to design software to utilize the increasing number of cores effectively.

The problem is that with an increasing number of cores, the efficiency of parallel software degrades gradually. Since matrix problems are fundamental to many scientific computing applications, we tested and benchmarked the performance of matrix factorizations on a shared-memory machine with 48 cores. Figure 1 shows the performance of Intel MKL [1], TBLAS [18], and PLASMA [3]. As shown in the figure, from one core to 48 cores, the efficiency (i.e., performance per core) of all three libraries decreases constantly.

At first glance, one would think it is the hardware or the operating system that imposes a limit to the parallel software. Indeed, this is true for I/O intensive applications due to limitations in I/O devices and critical sections in device drivers [6, 10, 21], but this argument is rarely true for computation-intensive applications. Computation-intensive applications typically perform in-memory computations and can minimize the overhead of system calls. By our intuitions, their performance should be scalable. However, it is not, as shown in Figure 1. Our paper studies why this unscalability problem can happen and how to solve the problem.

To solve the unscalability problem, we first want to find out what and where the bottlenecks are. We take QR factorization as an example and use PAPI [2] to collect a set of hardware performance counter data. By comparing experiments on a small number of cores to experiments on a large number of cores, we find the main causes for the performance degradation, which are "FPU idle cycles" and "Any stall cycles". However, there are many sources that can contribute to the causes. This leads us to conduct another set of experiments to identify the real reason. The reason we eventually find out is not a big surprise, but confirms one of our speculations. It is the increasing number of remote memory accesses that are incurred by the hierarchical NUMA (Non-Uniform Memory Access) architecture which is commonly found on modern manycore systems.

After finding the reason, we start to seek a solution to the problem. The main idea is as follows. First, at algorithm level, we extend the tile algorithms proposed by Dongarra et al. [8, 9] to design new NUMA-aware algorithms. Instead of

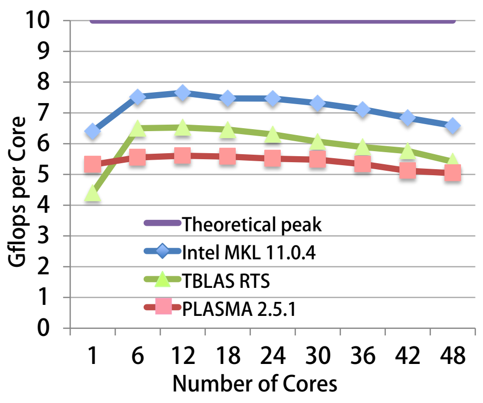


Figure 1: QR factorization on a 48-core AMD Opteron machine. With more cores, the *per-core performance keeps dropping.* Input sizes n $= 1000 \times$ NumberCores. One core does not reach the max performance because it is given a small input and has a relatively larger scheduling overhead.

allocating a global matrix and allowing any thread to access any matrix element, we restrict each thread to a fixed subset of the matrix. Since each thread has its own subset of data, a thread can view its own data as *local* and others as *remote*. The NUMA-aware algorithms take into account the number of accesses to remote data. For instance, if a thread needs to access remote data many times, it copies the remote data to its local NUMA node before accessing it. Second, we design a novel dynamic scheduling runtime system to support systems with a large number of CPU cores. The new runtime system follows a push-based data-flow programming model. It also uses work stealing to attain better load balancing. Third, we use a 2-D block cyclic distribution method [13] to map a matrix to different threads. Fourth, each thread owns a memory pool that is allocated from its local NUMA node and uses this memory to store its mapped subset of matrix.

Our experiment with QR factorizations demonstrates great scalability of the new approach. On a Linux machine with 48 cores, our performance per core drops only 3% from one core to 48 cores. This is an improvement of 16% over the previous work. We also test it on an extreme-scale shared-memory system with 1024 CPU cores. The system runs a single Linux operating system image. Based on the experimental results, we can deliver a maximum performance of 3.4 TFlops. It is 750% faster than the Intel MKL library, 75% faster than the Intel MKL ScaLAPACK library, and 42% faster than the PLASMA library [3] developed by the University of Tennessee.

To our best knowledge, this paper makes the following contributions:

1. Performance analysis of the unscalability issue in mathematical software for manycore systems, as well as an approach to solving the issue.

2. An extended matrix computation algorithm with NUMA awareness.

3. An extremely scalable runtime system with new mechanisms to effectively reduce NUMA memory accesses.

4. A novel framework that runs efficiently on manycore systems. It is the first time to demonstrate great scalability on shared-memory massively manycore systems with 1000 CPU cores.

The rest of the paper is organized as follows. Next section conducts performance analysis and identifies the reasons of the unscalability issue. Section 3 introduces our extended NUMA-aware QR factorization algorithm. Sections 4 and 5 describe the implementation of our framework. Section 6 provides the experimental results. Section 7 presents the related work. Finally Section 8 summarizes our work.

2. PERFORMANCE ANALYSIS

We measured the performance of an existing dynamic scheduling runtime system called TBLAS [18] to compute matrix operations. TBLAS was previously tested on quad-core and eight-core compute nodes, but has not been tested on compute nodes with more than sixteen CPU cores.

To measure TBLAS's performance on modern manycore systems, we did experiments with QR factorizations on a shared-memory 48-core machine. The 48-core machine has four 2.5 GHz AMD Opteron chips, each of which has twelve CPU cores. It also runs the CentOS 5.9 operating system and has 256 GB memory partitioned into eight NUMA nodes. Figure 1 displays performance of the QR factorization implemented with TBLAS. Each experiment takes as input a matrix of size $n = 1000 \times NumberCores$. On six cores, TBLAS reaches the highest performance of 6.5 GFlops/core. Then its performance keeps dropping. On 48 cores, its performance becomes as slow as 5.1 GFlops/core (i.e., a 20% loss).

However, we know that QR factorization has a time complexity of $O(\frac{4}{3}n^3)$ and is CPU-bound. With an increased input size, we had expected to see great scalability as previously shown on clusters with thousands of cores [18], because its ratio of communication to computation becomes less and less as the input size increases. Hence, Figure 1 exposes an unexpected unscalable problem. Our immediate task is to investigate what has caused this unscalability problem. Is it due to hardware or software or due to both?

2.1 Performance Analysis Using Hardware Performance Counters

We manually instrumented our matrix factorization program using the PAPI library [2]. Then we compared the differences between one experiment with a small number of cores and another experiment with 48 cores. Since this is a multi-threaded program, we collected performance data of hardware performance counters for each thread. Based on our performance data, each thread spent an equal amount of time on computation. This implies that there is no load imbalance among threads.

Table 1 shows the performance-counter data we collected on 12 and 48 cores, respectively. From the table, we can see that the 48-core experiment keeps the same low cache-miss rate and branch miss-prediction rate, as well as the same low rate of cycles without instructions. By contrast, the TLB miss rate increases a little bit from 7.8% to 8.7%. Moreover, the rate of FPU idle cycles increases by 31% from 3.9% to 5.1%. The rate of any-stall cycles (due to any resources) increases by almost 75%.

Table 1: Data of hardware performance counters collected from a 12-core experiment and a 48-core experiment, respectively.

	On 12 Cores	On 48 Cores
Performance per Core	6.4 GFlops	5.1 GFLops
L1 Data Cache Miss Rate	.5%	.5%
L2 Data Cache Miss Rate	2.98%	2.87%
TLB Miss Rate	7.8%	**8.7%**
Branch Miss-Prediction Rate	1.6%	1.5%
Cycles w/o Inst.	.5%	.6%
FPU Idle Cycles	3.9%	**5.1%**
Any-Stall Cycles	20.6%	**35.9%**

2.2 Reasons for Increased Any-Stall Cycles

We expect there are three possible reasons that can result in lots of stall cycles (due to any resources). They are: (1) thread synchronization, (2) remote memory access latency, and (3) memory bus or memory controller contention.

Thread synchronization can happen during the scheduling of tasks by the TBLAS runtime system. For instance, a thread is waiting to enter a critical section (e.g., a ready task queue) to pick up a task. When a synchronization occurs, the thread cannot do any computation but waiting. Therefore, the synchronization overhead is a part of the thread's non-computation time (i.e., total execution time - computation time). However, as shown in Table 2, the non-computation time of our program is less than 1%. Therefore, we can omit thread synchronization as a possible reason.

Table 2: Analysis of synchronization overhead.

	Total Time (s)	Computation Time (s)
12 Cores	30	29.9
48 Cores	74.8	74

The second possible reason is remote memory accesses. We conduct a different experiment to test the effect of remote memory accesses. In the experiment, the TBLAS program allocates memory only from the NUMA memory nodes from 4 to 7, that is, the second half of the eight NUMA memory nodes as shown in Figure 2. This is enforced by using the Linux command `numactl`. Then, we compare two configurations (see Figure 2): (a) running twelve threads on CPU cores from 0 to 11 located on NUMA nodes 0 and 1, and (b) running twelve threads on CPU cores from 36 to 47 located on NUMA nodes 6 and 7. In the first configuration, TBLAS attains a total performance of 57 GFlops. In the second configuration, it attains a total performance of 63 GFLOPS. Note that the matrix input is stored in NUMA nodes from 4 to 7. The performance difference shows that accessing remote memory does decrease performance. Note that we use Linux command `taskset` to pin each thread to a specific core.

The third possible reason is memory contention that may come from a memory bus or a memory controller. Here we do not differentiate the controller contention from the bus contention, but consider the memory controller an end point of its attached memory bus. In our third experiment, we run "numactl -i all" to place memory to all the eight NUMA nodes using the round-robin policy. As shown in Figure 3, there are two configurations: (a) we create twelve threads and pin them on cores from 0 to 11, (b) we create twelve thread and each thread runs on one core out of every four cores. In both configurations, each thread must access all the eight NUMA nodes due to the interleaved memory placement. Since the second configuration provides more memory bandwidth to each thread (i.e., lesser contention) than the first configuration, it achieves a total performance of 76.2 GFlops. By contrast, the first configuration only achieves a total performance of 66.3 GFlops. The difference shows that memory contention also has a significant impact on performance. One way to reduce memory contention is to maximize data locality and minimize cross-chip memory access traffic.

The above three experiments together with the performance differences they have made, provide an insight into the unscalability problem and motivate us to perform NUMA memory optimizations. We target at minimizing remote memory accesses since it is able to not only reduce the latency to access remote memories, but also alleviate the pressure on memory buses.

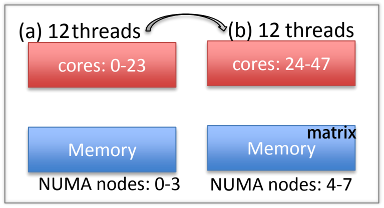


Figure 2: Effect of remote NUMA memory access latency. `matrix` is always allocated on NUMA nodes from 4 to 7. Then we run twelve threads in two different locations: (a) on NUMA nodes 0 and 1 (i.e., far from data), and (b) on NUMA nodes 6 and 7 (i.e., close to data).

3. THE TILE ALGORITHM

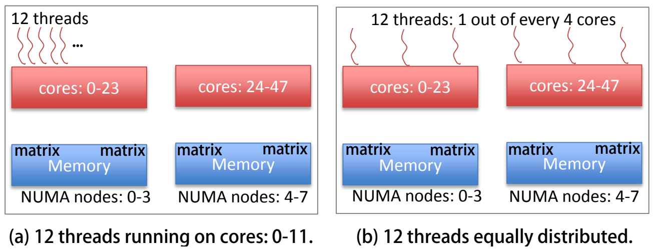


Figure 3: Effect of memory bus contention. `matrix` is allocated to all eight NUMA nodes using the roundrobin policy. We start twelve threads using two configurations, respectively. (a) 12 threads are located on 12 contiguous cores from 0 to 11. (b) 12 threads, each of which runs on one core out of every four cores.

To achieve the best performance, we must redesign new algorithms that can take advantage of both architecture and the application-level knowledges. We first briefly introduce the existing tile QR factorization algorithm. Then we analyze the algorithm's data-flow and data-reuse patterns. Finally we introduce our extension to make the algorithm NUMA aware.

3.1 Background

The tile QR factorization algorithm [8,9] uses an updating-based scheme that operates on matrices stored in a tile data layout. A tile is a block of submatrix and is stored in memory contiguously. Given a matrix A consisting of $n_b \times n_b$ tiles, matrix A can be expressed as follows:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,n_b} \\ A_{2,1} & A_{2,2} & \dots & A_{2,n_b} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n_b,1} & A_{n_b,2} & \dots & A_{n_b,n_b} \end{pmatrix},$$

where $A_{i,j}$ is a square tile of size $b \times b$.

At the beginning, the algorithm computes the QR factorization for tile $A_{1,1}$ only. The factorization output of $A_{1,1}$ is then used to update the set of tiles on $A_{1,1}$'s right hand side in an embarrassingly parallel way (i.e., $\{A_{1,2}, \dots, A_{1,n_b}\}$). As soon as the update on any tile $A_{1,j}$ is finished, the update on tile $A_{2,j}$ can read the modified $A_{1,j}$ and start. In other words, whenever a tile-update in the i-th row completes, its below tile in the $(i+1)$-th row can start if $A_{i+1,1}$ also completes. After updating the tiles in the bottom n_b-th row, tile QR factorization applies the same steps to the trailing submatrix $A_{2:n_b,2:n_b}$ recursively.

3.2 Algorithm Analysis

Figure 4 shows the data-flow graph for solving a matrix of 4 × 4 tiles using the tile QR factorization algorithm. In the figure, each node denotes a computational task. A task is represented by a function name and an index [i, j]. A task with index [i, j] indicates that the output of the task is tile A[i, j]. An edge from node x to node y indicates that the output of node x will be an input to node y. For instance, `SSRFB` [2, 4] refers to a task that reads the output of tasks `LARFB` [1, 4] and `TSQRT` [2, 1], then modifies its own output of A[2, 4].

There are totally four functions (aka "kernels") in the tile QR factorization algorithm: `GEQRT`, `TSQRT`, `LARFB`, and `SSRFB`. For completeness of this paper, we describe them briefly here:

- `GEQRT`: R[k,k], V[k,k], T[k,k] ← GEQRT(A[k,k]).

  It computes the QR factorization for a tile A[k,k] located on matrix A's main diagonal, and generates three outputs: an upper triangular tile R[k,k], a unit lower triangular tile V[k,k] containing the Householder reflectors, and an upper triangular tile T[k,k] for storing the accumulated transformations.

- `TSQRT`: R[k,k], V[i,k], T[i,k] ← TSQRT(R[k,k], A[i,k]).

  It updates the tiles located under tile A[k,k]. After GEQRT is called, TSQRT stacks tile R[k,k] on top of tile A[i,k] and computes an updated factorization.

- `LARFB`: R[k,j] ← LARFB(V[k,k], T[k,k], A[k,j]).

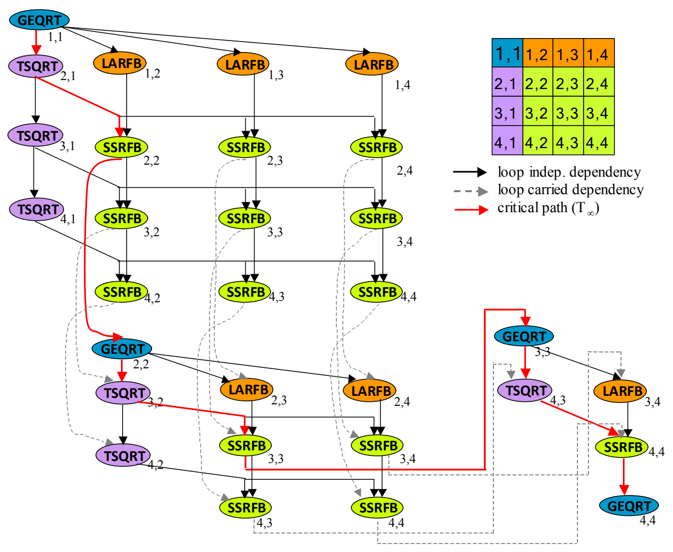


Figure 4: Data flow graph of the tile QR factorization algorithm to solve a matrix of 4×4 tiles.

  It updates the tiles located on the right hand side of A[k,k]. LARFB applies GEQRT's output to tile A[k,j] and computes the R factor R[k,j].

- `SSRFB`: R[k,j], A[i,j] ← SSRFB(V[i,k], T[i,k], R[k,j], A[i,j]).

  It updates the tiles located in the trailing submtrix of $A_{k+1:n_b,k+1:n_b}$. It applies TSQRT's output to a stacked R[k,j] and A[i,j], and updates them correspondingly.

It is important to note that Figure 4 provides not only the information of data dependency, but also the information of data reuse. The previous TBLAS runtime system and many other runtime systems only use the data dependency information to ensure correct computational results.

Data reuse is the other important information to be utilized. It indicates how many times a tile will be used by the other tasks. For instance, in Figure 4, suppose tasks of `SSRFB` [2,2], [2,3], [2,4] are computed by the same thread running on a NUMA node p, and task `TSQRT` [2,1] is computed by another thread running on a different NUMA node q. Then due to cache misses, the former thread has to read the remote tile A[2,1] three times. Note that on NUMA systems, the latency to access a remote NUMA node can be much higher than that to access a local NUMA node (e.g., twice higher on our 48-core AMD machine). Therefore, we modify the tile algorithm and the TBLAS runtime system to utilize the data-reuse information to optimize NUMA memory accesses.

3.3 A NUMA-Aware Extension

This subsection describes our extended algorithm to solve QR factorizations by minimizing remote NUMA memory accesses.

We assume that the input to the algorithm is a matrix A with $nb \times nb$ blocks. Given a number of n threads, the $nb \times nb$ blocks are mapped to the n threads using a static dis-

Algorithm 1 Extended_NUMA_Tile_QR Algorithm

```
Thread_init(int tid /*thread id*/, double A[nb][nb])
double* $A_{tid}$[nb][nb]
for i ← 0 to nb-1 do
  for j ← 0 to nb-1 do
    /*Each tile[i,j] is mapped to a thread*/
    if get_assigned_thread(i, j) = tid then
      $A_{tid}$[i][j] ← malloc(tile_size)
      $A_{tid}$[i][j] ← A[i][j];
    end if
  end for
end for


Thread_entry_point(int tid)
for k ← 0 to nb-1 do
  if get_assigned_thread(k, k) = tid then
    $R_{tid}$[k,k], $V_{tid}$[k,k], $T_{tid}$[k,k] ← geqrt($A_{tid}$[k,k])
    [memcpy $V_{tid}$[k,k], $T_{tid}$[k,k] to $V_{tid_x}$[k,k], $T_{tid_x}$[k,k]
    in threads $tid_x$, where thread $tid_x$ is waiting for
    $V_{tid}/T_{tid}$[k,k]]
  end if
  thread_barrier
  for j ← k+1 to nb-1 /*along k-th row*/ do
    if get_assigned_thread(k, j) = tid then
      $A_{tid}$[k,j] ← larfb($V$[k,k], $T$[k,k], $A$[k,j])
      [memcpy $A_{tid}$[k,j] to $A_{tid'}$[k,j], where thread $tid'$ is
      waiting for $A_{tid}$[k,j]]
    end if
  end for
  thread_barrier
  for i ← k+1 to nb-1 do
    if get_assigned_thread(i, k) = tid then
      $R$[k,k], $V$[i,k], $T$[i,k] ← tsqrt($R$[k,k], $A$[i,k])
      [memcpy $V_{tid}$[i,k], $T_{tid}$[i,k] to $V_{tid_x}[i,k]$, $T_{tid_x}$[i,k]
      in threads $tid_x$, where thread $tid_x$ is waiting for
      $V_{tid}/T_{tid}$[i,k]]
    end if
    for j ← k+1 to nb-1 do
      if get_assigned_thread(i, j) = tid then
        $A_{tid}$[k,j], $A_{tid}$[i,j]←ssrfb($V_{tid}$[i,k], $T_{tid}$[i,k],
        $A_{tid}$[k,j], $A_{tid}$[i,j])
        [memcpy $A_{tid}$[k,j] to $A_{tid'}$[k,j], where thread $tid'$
        is waiting for $A_{tid}$[k,j]]
      end if
    end for
    thread_barrier
  end for
end for
```

tribution method. The static distribution method is defined by the means of an application-specific mapping function `get_assigned_thread(i,j)` which calculates the mapped thread ID of a matrix block [i, j].

Each thread has a thread ID *tid*. It uses its local NUMA memory to store its assigned subset of the input matrix A denoted by A_{tid}, and intermediate matrix results denoted by V_{tid} and T_{tid}. Those thread-private matrices of A_{tid}, V_{tid}, and T_{tid} are implemented as $nb \times nb$ NULL pointers. Their memories are dynamically allocated when needed.

Algorithm 1 shows the multithreaded version of NUMA-aware tile QR factorization. Before starting computation, each thread *tid* calls `Thread_init` to copy input to its private A_{tid}, from either a global matrix or a file in parallel. Next, each thread computes QR factorization in parallel as follows: 1) factor block A[k,k] on the main diagonal by calling `geqrt` if A[k,k] is assigned to itself; 2) factor all blocks on the k-th row by calling `larfb`; 3) factor the remaining matrix blocks from A[k+1,k] to A[nb-1,nb-1] by calling `tsqrt` and `ssrfb` in parallel. Note that the algorithm copies certain tasks' output to a set of remote threads explicitly. The set of threads that are waiting for the specific output can be determined by the data flow analysis of the algorithm as shown in Figure 4.

The purpose of `thread_barrier` is only to show a correct working version of the parallel algorithm. We acknowledge that a direct translation of the algorithm may not be the fastest implementation due to the barriers. In our own implementation using a runtime system, however, there is no barrier at all. We use a dynamic data-availability-driven scheduling scheme in the runtime system to trigger new tasks whenever their inputs become available and are able to avoid global synchronizations. As shown in Figure 4, a runtime system can execute any task as long as its inputs are ready. A ready task can be scheduled to start even though some tasks prior to it are blocked. This approach essentially uses the same idea as the out-of-order instruction execution in superscalar processors.

4. OVERVIEW OF OUR FRAMEWORK

To support manycore NUMA systems efficiently, we design a framework that integrates application, runtime system, and architecture together. The framework consists of a static distribution method, an architecture-aware runtime system, and a user program. The user program is used to create (or "eject") new tasks which will be scheduled dynamically by the runtime system.

Our framework adopts a data-centric computing model. It co-allocates both data and tasks to different threads using a static distribution method. A matrix block A[i, j] is first assigned to a thread. Next, the task whose output is block A[i, j] will be assigned to the same thread as the block A[i, j]. On shared-memory systems, the static distribution will not prevent a thread from accessing any data. However, by using the static distribution, we are able to allocate all of a thread's data from its local NUMA memory and minimize remote memory accesses.

A user program is executed by the runtime system. Whenever reading a computational function, the runtime system creates a task and puts it to a queue immediately without doing any computation (i.e., "eject" a task or a job). A created task contains all the information needed for a thread to execute such as function name, locations of input and output. There are two types of task pool: waiting-task pool and ready-task pool. When a new task is created, it first enters the waiting-task pool. After all of its inputs are ready, it becomes a ready task and moves to a ready task pool.

The entire framework follows a data-flow programming model. It drives the user-program execution by data availability. After finishing a task and generating a new output, a compute thread goes to the waiting-task pool to search for the tasks whose input is the same as that output. If a waiting task has the same input as the newly generated output, its input state changes from unready to ready. When all the input states change to ready, the waiting task becomes a ready task and will be later picked up by a compute thread. Note that each compute thread is independent from every other compute thread (i.e., there is no fork-join).

In order to minimize remote NUMA memory accesses, our runtime system copies an output to the waiting threads' local memory nodes. The memory copy operation happens before the input state changes (from unready to ready) so that the local copy of the output can be used by the waiting thread.

In general, each compute thread executes the following steps in a while loop: (1) picks up a ready task, (2) computes the task, (3) searches for tasks that are waiting for the new output, and (4) triggers new ready tasks. Whenever a task is finished, the runtime system will use the newly available output data to trigger a set of new tasks.

5. THE IMPLEMENTATION

Our work builds upon the previous TBLAS runtime system to support scalable matrix computations on shared-memory manycore systems.

5.1 TBLAS

The TBLAS runtime system was originally designed to support matrix computations on distributed multicore cluster systems [18]. Although TBLAS has demonstrated good scalability on clusters, it has been applied only to compute nodes with a maximum number of 12 CPU cores. The structure of TBLAS is very simple. It has one task window and a global ready task queue shared by all the compute threads. The task window is of fixed size and stores all the waiting task (i.e., tasks created but not finished). The ready queue stores all the ready tasks whose input are available to read. The runtime system includes two types of thread: task-generation thread and compute thread. Each CPU core has a compute thread, but the entire runtime system has a single task-generation thread. The task-generation thread executes the user program and create new tasks to fill in the task window. Whenever becoming idle, a compute thread picks up a ready task from the global ready task queue. After finishing the task, the compute thread searches for the finished task's children and triggers new tasks.

5.2 Extensions

To achieve scalability on a shared-memory system with hundreds or even thousands of cores and a complex NUMA memory hierarchy, we have extended the previous TBLAS runtime system in the following aspects. Suppose a shared-memory system has n CPU cores and m NUMA memory nodes, our runtime system will launch n compute threads on n different cores.

- **Static data and tasks partitioning**. A matrix with $n_b \times n_b$ tiles will be partitioned among different threads using a 2-D cyclic distribution method [13]. Assume that n threads are mapped to a 2-D grid of r rows and c columns, where $n = r \times c$. Given a tile A[i, j] that is located at the i-th row and j-th column of matrix A, tile A[i, j] will be assigned to thread[i mod r, j mod c]. This distribution method is called 2-D cyclic distribution. All the matrix tiles assigned to thread i will be stored to thread i's local NUMA memory node. Also, a task whose output is tile A[i, j] will be assigned to the same thread as A[i, j]. This way a thread can always perform write operations to its local NUMA memory.

- **Per-thread ready queue**. As shown in Figure 5, instead of using a global ready task queue, each thread has its own ready task queue. A thread's ready task queue only stores the tasks that are assigned to it based on the 2-D cyclic distribution. If a task modifies a tile that is assigned to thread i, the task is added to thread i's ready task queue accordingly. Using a per-thread ready queue not only reduces thread contention but also increases data locality.

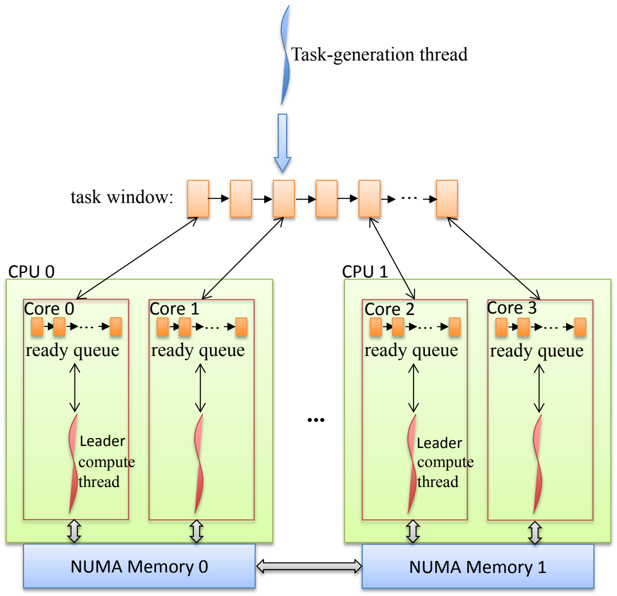


Figure 5: Architecture of the new TBLAS runtime system designed for NUMA many cores. For simplicity, the figure only displays two dual-core CPUs and two NUMA nodes. The actual runtime system can support many cores and many NUMA nodes.

- **Thread-private memory allocator**. Each thread allocates memory from its private memory pool to store its assigned submatrix and intermediate results. Before a thread starts, it has pre-allocated a slab of memory for its memory pool. When the thread starts, the first thing it does is to touch all the pages associated with the memory pool. This way all physical memory frames will reside in the thread's local NUMA memory node.

- **Work stealing**. Since each thread is assigned a fixed subset of tasks based on the 2-D cyclic distribution method, it results in a suboptimal performance due to load imbalance. To obtain better load balancing, our runtime system allows each thread to do work stealing when a thread has been idle for a certain amount of time. Note that a thread just steals tasks, but not the tasks' input or output data even if the data are in a remote NUMA node. We have implemented two work stealing policies: 1) locality-based work stealing, where a thread first tries to steal tasks from its neighbors that are on the same NUMA node, then steals from other threads located on remote NUMA nodes; 2) system-wise random work stealing, where a thread steals tasks from any other thread randomly.

- **Inter-NUMA-node memory copy**. If a matrix block owned by thread x is accessed multiple times by a remote thread y, the runtime system calls `memcpy` to copy the block to thread y's local buffer. After `memcpy` is complete, thread y can access the block from its local buffer directly.

Although the output block of a task is stored to a buffer in the destination, it will not stay in the destination forever since our runtime system keeps track of a reference counter for each block. After the consumers read (or consume) the block for a certain number of times, the block will be freed to the runtime system's free-memory pool and can be reused for future memory copies. Therefore, this NUMA optimization method has good scalability in terms of memory usage.

In our current implementation, the algorithm specifies which block should be copied to remote NUMA nodes. Note that given any matrix block, the runtime system knows to which thread the block is assigned by the predefined static 2-D cyclic distribution method.

- **NUMA-group leader thread**. We use leader threads to further reduce the cost of memory copies. Figure 5 illustrates that each NUMA node has a group of compute threads (one thread per core). The thread that runs on the first core of the NUMA node is defined as the "leader thread" of the group. The leader thread has a larger memory pool than its group members. When a data block is needed by multiple threads that reside on a remote NUMA node, the runtime system copies the data block to that NUMA node's leader thread only once. After the block is copied to the remote leader thread, member threads on the remote NUMA node can read data from their leader correspondingly. Note that all accesses from the member threads to their leader are local memory accesses, which saves a lot of remote memory accesses.

We use the above enhancements to attain better locality, load balancing, scalability, reduced synchronizations, and NUMA optimization. In addition, without the support of the runtime system, it will be much more difficult for programmers to manage and copy memory around manually to optimize NUMA memory accesses.

6. EXPERIMENTAL RESULTS

This section presents experimental results with QR factorizations on two *shared-memory* multicore systems: a system with 48 CPU cores, and a system with 1024 CPU cores.

6.1 Evaluation on a 48-core System

The 48-core system is the same one as we used for the performance analysis in Section 2. It has four 12-core AMD Opteron CPUs. Each CPU chip consists of two packages each of which has a dedicated NUMA memory node. Hence the system has eight NUMA memory nodes.

In the following experiments, we choose the appropriate NUMA memory placements to obtain the best performance. For instance, in the single CPU experiment, we use the Linux command "numactl -m node" to allocate memory to the CPU's local NUMA node. And in the largest 48-core experiment, we used "numactl -i all" to interleave allocation to all the NUMA nodes. Note that the performance will become much worse if numactl is not used.

6.1.1 *Initial Optimizations*

The previous TBLAS runtime system does not have good scalability. Figure 6 shows that the previous TBLAS on a single CPU (i.e. 6 cores) attains a performance of 6.5 GFlops/core. With an increasing number of cores, its performance starts to decrease gradually. Eventually on 48 cores, it drops to 5.4 GFlops/core (i.e., 17% loss). Note that the input matrix is of size $N = 1000 \times$ NumberCores.

The first optimization we apply is to divide a global ready-task queue into multiple queues so that each thread has a private ready task queue. This optimization can reduce thread contention to enter the global queue. We use the 2-D cyclic distribution method to distribute tasks to different threads statically as described in Section 5.2. As shown in Figure 6, adding the 2-D cyclic distribution makes the program scale better (i.e., a more constant GFlops/core). However, on a small number of cores, the previous TBLAS is faster than our extended 2-D cyclic version. This is because the dynamic scheduling method of the previous TBLAS has a better load balancing than the static distribution method. Furthermore, the thread contention overhead is low when only six threads are used.

The second optimization is to use work stealing (WS) to improve load balancing. Figure 6 shows that adding work stealing improves the static 2D cyclic extension by another 6%. When the number of cores is greater than 24, the `2D cyclic + WS` extension starts to outperform the previous TBLAS system.

However, from 24 cores to 48 cores, `2D cyclic + WS` still drops gradually. This shows that load balancing across different cores and using block-based data layout are not sufficient. Based on the insight provided by our performance analysis (Section 2), we start to perform NUMA memory optimizations.

6.1.2 *Adding NUMA Optimizations*

In order to minimize remote memory accesses, we not only distribute *tasks* to different threads statically, but also distribute *data* to different threads statically. The location of a task and the location of the task's output are always the same. This feature is enforced by the 2-D cyclic distribution method.

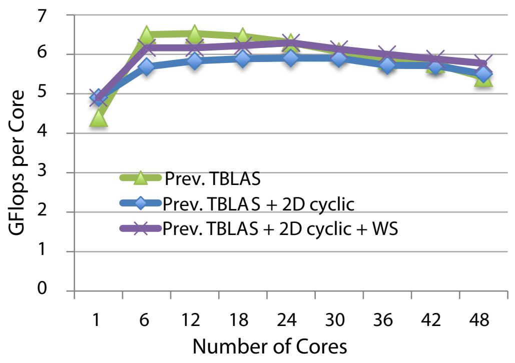


Figure 6: Our initial attempt to add 2-D cyclic distribution and work stealing to the previous TBLAS runtime system. The modified versions have a better scalability but are slower when using a small number of cores.

We have implemented the NUMA-aware QR factorization using our new TBLAS runtime system. The NUMA-aware algorithm decides which matrix block should be copied to remote NUMA memory nodes. Then the runtime system will take care of all the remaining work. For instance, the runtime system first determines which threads are waiting for the block, then pushes the block to the consumer threads' local memory buffers, finally it notifies the consumer threads of the data availability. The major NUMA optimizations added to the TBLAS runtime system are: 1) static partitioning and co-allocation of data and tasks, 2) automatic inter-NUMA memory copy, 3) NUMA-group leaders, 4) thread-private and group-private memory allocators, and 5) locality-aware work stealing. Please refer to Section 5.2 for details.

Figure 7 shows the performance of our new TBLAS that has added the NUMA optimizations. Without work stealing, the new TBLAS runtime system already shows great scalability. For instance, on 18 cores, it attains a maximum performance of 6.3 GFlops/core. On all the 48 cores, it attains 6.1 GFlops/core (i.e., 3% less than the maximum). By enabling work stealing, we are able to further improve the performance slightly (up to 4%). The figure also displays the performance of PLASMA 2.5.1 as a reference. To get the best performance of PLASMA, we have run its experiments with numactl, chosen the static scheduler, and tried various tile sizes. The performance difference between the previous TBLAS and PLASMA is due to the load-balance difference between dynamic scheduling and static scheduling.

It is worthwhile to point out that the new TBLAS with work stealing can perform as well as the previous TBLAS that runs on a single local memory node (i.e., no remote memory accesses). In addition, it is 16% faster than the previous TBLAS when using 48 cores.

6.2 Evaluation on a Massively Manycore System with 1024 Cores

The largest shared-memory manycore system we have access is an SGI UV1000 system [17]. It has 1024 CPU cores (Intel Xeon X7550 2.0GHz), and runs a single operating system image of Red Hat Enterprise Linux 6. It has 4 Terabytes of global shared memory distributed across 64 blades. Each blade consists of two 8-core CPUs and two NUMA memory nodes. All the blades are connected by a NUMAlink 5 interconnection.

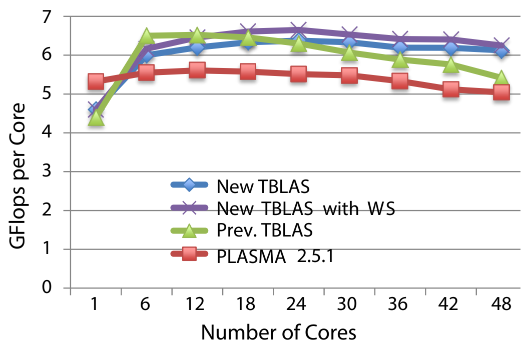


Figure 7: Performance of the new TBLAS runtime system with NUMA optimizations on a 48-core AMD Opteron system.

We run experiments on the SGI UV1000 system using one core to 960 cores. The input matrix size $N = 2000 \times \sqrt{number\_cores}$. As shown in Figure 8, we compare five different programs. To make a fair comparison, we have tuned each program to obtain its best performance. We describe the five programs as follows:

- `Intel MKL (LAPACK) QR`. We use Intel MKL 10.3.6 to compute QR factorizations using its LAPACK routine. It creates one process with a number of threads to perform computations. We run the Linux command "numactl –interleave=memory-nodes" to execute the program.

- `PLASMA QR`. PLASMA is a parallel multithreading library for multicore systems. We test various tile sizes and choose the size that provides the best performance. We also use PLASMA's static scheduler and "numactl –interleave=memory-nodes" to run the experiments.

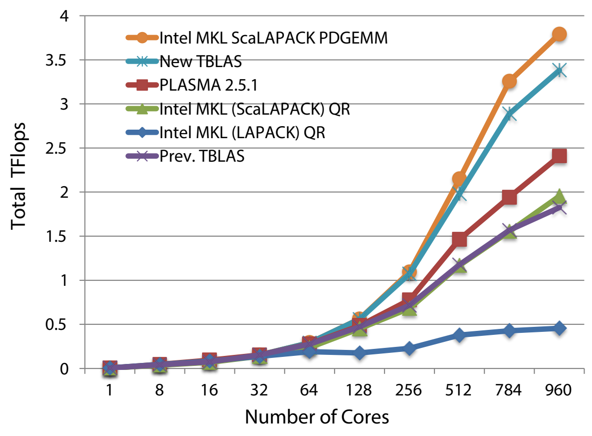
- `New TBLAS`. Because the new TBLAS runtime system itself manages memory allocation and memory copy to optimize NUMA accesses, we use "numactl –localalloc" to allocate memory on the local NUMA memory node. We tune its tile size to achieve the best performance.

- `Intel MKL (ScaLAPACK) QR`. Intel MKL provides two sets of computational routines: *shared-memory* LAPACK routines, and *distributed-memory* MPI-based ScaLAPACK routines. Since the memory on the SGI UV1000 system is distributed to various blades, another interesting experiment is to test the ScaLAPACK routine which uses the MPI message passing model.

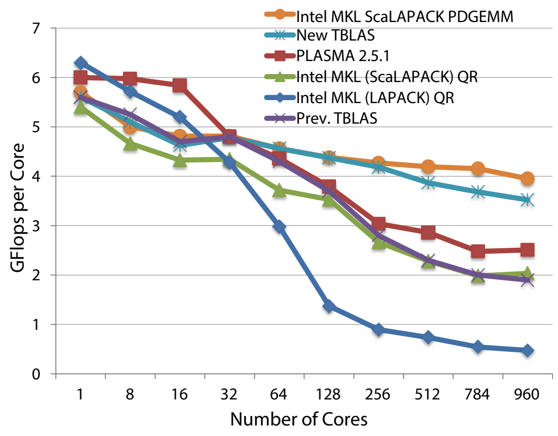
  We tune three optimization parameters for the ScaLAPACK QR factorization: i) selection of one-process-per-node or one-process-per-core, ii) ScaLAPACK block size NB, iii) process grid $P \times Q$, where the total number of processes is equal to $P \times Q$. To attain the highest performance, we choose one-process-per-core, NB=100, and the $P^* \times Q^*$ with the best performance.

- `Intel MKL ScaLAPACK PDGEMM`. PDGEMM is a parallel *matrix multiplication* subroutine in ScaLAPACK. The reason we compare QR factorizations to PDGEMM is that matrix multiplication has a higher computation-to-communication ratio, a higher degree of parallelism, and faster kernel implementations than the QR factorization. Therefore, we can use PDGEMM as an upper bound for QR factorizations. In fact, matrix multiplication can serve as an upper bound for most matrix computations although it may not be a tight upper bound.

Figure 8 (a) shows that our new TBLAS is able to keep up with the speed of PDGEMM, and scales efficiently from one core to 960 cores. Although the SGI UV1000 system has 1024 cores, we were not able to reserve all of them due to long job waiting time and hardware issues. The new TBLAS program delivers 1.05 TFlops, 2.0 TFlops, and 2.9 TFlops on 256, 512, and 784 cores, respectively. On 960 cores, it can deliver 3.4 TFlops. Note that the speedup of PDGEMM slows down a little bit from 784 to 960 cores. We think it might hit some hardware limitation because parallel matrix multiplication typically has a perfect weak-scalability.



(a) Overall performance.



(b) Performance per core.

Figure 8: Comparison of the new TBLAS QR implementation with Intel MKL LAPACK, Intel MKL ScaLAPACK, PLASMA, previous TBLAS, and parallel matrix multiplications (i.e., PDGEMM) on a shared memory 1024-core SGI UV1000 system.

PLASMA is the second best library, which can deliver 2.4 TFlops using 960 cores. As shown in Figure 8 (b), when the number of cores is less than or equal to 16 (i.e., within one blade), new TBLAS is slower than PLASMA because we disabled work stealing for better performance on large numbers of cores. On the other hand, the Intel MKL 10.3.6 library achieves 0.45 TFlops by using the MKL LAPACK routine, and 1.95 TFlops by using the MKL ScaLAPACK routine, respectively. Figure 8 shows that by taking into account NUMA optimizations, our new TBLAS can outperform its counterpart of Intel MKL LAPACK by 750%, and the ScaLAPACK library by 75%.

7. RELATED WORK

There are a few mathematical libraries that can provide high performance on multicore architectures. Intel MKL, AMD ACML, and IBM ESSL are vendor-provided libraries. They support linear algebra computations as one fundamental area. All the linear algebra subroutines have implemented the standard LAPACK interface [4]. PLASMA is an open source linear algebra library developed at the University of Tennessee [3]. It is a reimplementation of LAPACK and aims at delivering high performance on multicore architectures. Similar to PLASMA, our work also uses the tile algorithms but extends the algorithms with NUMA optimizations. ScaLAPACK is a high performance linear algebra library for distributed memory machines [5]. ScaLAPACK uses a message passing programming model. Different from ScaLAPACK, our work focuses on how to design scalable software using a shared-memory programming model.

While the existing libraries can utilize all CPU cores, their efficiency (e.g., GFlops per core) keeps dropping when the number of cores increases. This paper targets at this unscalability problem and proposes an approach to improving it. We compare our framework with the most widely used Intel MKL and the latest PLASMA library and demonstrate better scalability.

NUMA-related issues on multicore systems have been studied by many researchers in the high performance computing community. McCurdy and Vetter introduced a tool called *Memphis*, which uses instruction-based sampling to pinpoint NUMA problems in benchmarks and applications [16]. Tao et al. designed a low-level hardware monitoring infrastructure to identify remote NUMA memory inefficiencies and used the information to specify proper data layout to optimize applications [20]. Li et al. used shared memory to implement NUMA-aware intra-node MPI collectives and have achieved twice speedup over traditional MPI implementations [14]. Tang et al. proposed a two-phase methodology to investigate the impact of NUMA on Google's large workloads and designed load-test to optimize NUMA performance [19]. Frasca et al. improved both CPU performance and energy consumption significantly for large sparse-graph applications by using graph reorganization and dynamic task scheduling to optimize NUMA performance [12].

NUMA issues have also been studied by many researchers in the operating system community. Majo and Gross investigated the NUMA-memory contention problem and developed a model to characterize the sharing of local and remote memory bandwidth [15]. Fedorova et al. designed a contention-aware algorithm *Carrefour* to manage memory traffic congestion in the Linux OS [11]. An extension of the GNU OpenMP, *ForestGOMP* [7], allows developers to provide hints to the runtime system to schedule thread and memory migrations. These system related researches mainly studied how and when to migrate thread and memory with a minimal cost. Differently, we take advantage of the domain-specific knowledge (i.e., matrix computations) and use a combination of 2-D cyclic distribution and work stealing to keep load balancing and minimize thread and memory movement.

8. CONCLUSION AND FUTURE WORK

Despite the increase in the number of cores per chip, parallel software has not been able to take the full advantage of all the cores and to scale efficiently on shared-memory manycore systems. We analyzed the performance of existing software and proposed a new framework that consists of an extended NUMA-aware algorithm and a new runtime system. The framework builds upon a static 2-D cyclic distribution such that tasks are co-located with their accessed data. If a data block is referenced by remote threads multiple times, the runtime system copies the data from a local memory node to a remote memory node. The runtime system uses a data-availability-driven (i.e., data-flow) scheduling method and knows where data are located and which tasks are waiting for the data. This way it is feasible to push data to consumers automatically. The experimental results on a 48-core system and a 1024-core system demonstrate that our approach is effective, and is much more scalable than the existing numerical libraries (e.g., up to 750% faster than Intel MKL on 960 cores).

While our approach is used to solve QR factorization as an example, it can be applied directly to solve other different matrix factorizations, systems of linear equations, and eigenvalue problems. We envision that the same methodology and principles can also be extended to support other types of scientific applications such as computational fluid dynamics, sparse matrix problems, and quantum chemistry. Our future work is to apply the framework to solve other matrix problems, and then extend it to support computational fluid dynamics applications and sparse matrix problems on distributed-memory manycore systems at extreme scale.

9. REFERENCES

[1] Intel Math Kernel Library (MKL). http://software.intel.com/en-us/intel-mkl. Accessed: January, 2014.

[2] PAPI project. http://icl.cs.utk.edu/papi. Accessed: January, 2014.

[3] PLASMA project. http://icl.cs.utk.edu/plasma. Accessed: January, 2014.

[4] E. Anderson. *LAPACK Users' Guide*, volume 9. SIAM, 1999.

[5] L. S. Blackford. *ScaLAPACK User's Guide*, volume 4. SIAM, 1997.

[6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, pages 1–8. USENIX, 2010.

[7] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier. Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 79–92. Springer, 2009.

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

[9] 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.

[10] B. Cantrill and J. Bonwick. Real-world concurrency. *ACM Queue*, 6(5):16–25, 2008.

[11] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *18th international conference on architectural support for programming languages and operating systems*. ACM, 2013.

[12] M. Frasca, K. Madduri, and P. Raghavan. NUMA-aware graph mining techniques for performance and energy efficiency. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 95:1–95:11. IEEE, 2012.

[13] 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, Sept. 1994.

[14] S. Li, T. Hoefler, and M. Snir. NUMA-aware shared-memory collective communication for MPI. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 85–96. ACM, 2013.

[15] Z. Majo and T. R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*. ACM, 2011.

[16] C. McCurdy and J. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 87–96. IEEE, 2010.

[17] SGI. *Technical Advances in the SGI UV Architecture (white paper)*. 2012.

[18] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*, pages 19:1–19:11. ACM, 2009.

[19] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *Nineteenth International Symposium on High-Performance Computer Architecture*. IEEE, 2013.

[20] J. Tao, W. Karl, and M. Schulz. Memory access behavior analysis of NUMA-based shared memory programs. *Sci. Program.*, 10(1):45–53, Jan. 2002.

[21] D. Waddington, J. Colmenares, J. Kuang, and F. Song. KV-Cache: A scalable high-performance web-object cache for manycore. In *Proceedings of the 6th ACM/IEEE International Conference on Utility and Cloud Computings*, UCC 2013. ACM, 2013.