SANDIA REPORT

SAND2016-9613
Unlimited Release
Printed September, 2016

Hierarchical Task-Data Parallelism using Kokkos and Qthreads

H. Carter Edwards, Stephen L. Olivier, Greg E. Mackey, Kyungjoo Kim, Michael Wolf, George W. Stelle, Jonathan W. Berry, Sivasankaran Rajamanickam

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

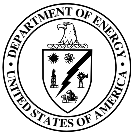
Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online



SAND2016-9613
Unlimited Release
Printed September, 2016

Hierarchical Task-Data Parallelism using Kokkos and Qthreads

H. Carter Edwards, *Scalable Algorithms*
Stephen L. Olivier, *Scalable System Software*
Greg E. Mackey, *Cyber Initiatives*
Kyungjoo Kim, *Computational Mathematics*
Michael Wolf, *Scalable Algorithms*
George W. Stelle, *Scalable System Software*
Jonathan W. Berry, *Discrete Math & Optimization*
Sivasankaran Rajamanickam, *Scalable Algorithms*
Sandia National Laboratories
P.O. Box 5800 / MS 1318
Albuquerque, NM 87185

Abstract

This report describes a new capability for hierarchical task-data parallelism using Sandia's Kokkos and Qthreads, and evaluation of this capability with sparse matrix Cholesky factorization and social network triangle enumeration mini-applications. Hierarchical task-data parallelism consists of a collection of tasks with executes-after dependences where each task contains data parallel operations performed on a team of hardware threads. The collection of tasks and dependences form a directed acyclic graph of tasks – a *task DAG*. Major challenges of this research and development effort include: portability and performance across multicore CPU; manycore Intel Xeon Phi, and NVIDIA GPU architectures; scalability with respect to hardware concurrency and size of the task DAG; and usability of the application programmer interface (API).

Acknowledgment

This work was funded by the Sandia National Laboratories' Laboratory Directed Research and Development (LDRD) program in the Computer and Information Science investment area.

We appreciate NVIDIA providing pre-release access to their nvcc v8.0 compiler and linker, which was critical for porting to NVIDIA's GPU architecture.

Contents

1 Introduction **13**

1.1 Concepts and Terminology 14

1.2 Kokkos / Qthreads Integration 16

1.3 Mini-Applications 16

2 Dynamic Task DAG Abstractions, Requirements, and Lessons Learned **17**

2.1 Life-cycle of a Task in a Dynamic Task DAG 17

2.1.1 An Abstract Perspective 17

2.1.2 An Illustrative Code Perspective 19

2.2 Lessons Learned and Evolving Abstractions 21

2.2.1 Spawning and Dependences 21

2.2.2 Compatibility of Serial and Thread Team Tasks 22

2.2.3 Memory Management – Thread Scalability 22

2.2.4 Memory Management – Task Scalability 22

2.2.5 Memory Management – Task Lifetime 23

2.2.6 Critical Paths and Short Paths 24

2.2.7 CUDA Porting 24

2.2.8 GPU Thread Team 24

2.3 Final Peer Review – Nomenclature Revisions 25

3 Task Schedulers and Runtime Systems **27**

3.1 Qthreads Back-end 27

3.2 Prototype Kokkos Serial, Pthreads, and CUDA implementations 29

3.3 Final Kokkos Serial, OpenMP, and CUDA implementations . . . 29
3.3.1 Managing Task DAG Queues . . . 30
3.3.2 Executing a Task DAG . . . 33
3.4 Portable and Thread Scalable Memory Pool . . . 38
3.4.1 Design and Implementation . . . 39
3.4.2 Superblock States . . . 41
3.4.3 Allocation . . . 44
3.4.4 Switching Superblocks . . . 45
3.4.5 Deallocation . . . 46
3.4.6 Empty . . . 46

4 Integration with the Multithreaded Graph Library (MTGL) 49
4.1 Introduction to the MTGL . . . 49
4.2 Graph abstractions in the MTGL . . . 50
4.3 The MTGL API . . . 52
4.3.1 Sizes, descriptors, and iterators . . . 52
4.3.2 Atomics and concurrency . . . 53
4.3.3 Generic functions . . . 53
4.3.4 Basic functionality . . . 54
Basic MTGL include files . . . 54
Starting out: typedefs and graph initialization . . . 54
Declaring iterators and property maps . . . 55
Initialization and use of property maps . . . 56
Device computations on property maps . . . 56
4.4 MTGL library algorithms . . . 58
4.4.1 Breadth-First Search . . . 59
4.4.2 Connected Components . . . 60

4.5 MTGL/Kokkos algorithm performance . . . 61
4.5.1 Basic array processing . . . 61
4.5.2 Graph Iteration . . . 62
4.5.3 Graph algorithm performance . . . 62
4.6 Customizing MTGL algorithms to optimize Kokkos usage . . . 64
4.7 Processing graphs that don't fit on device . . . 65

5 Tacho: Sparse Incomplete Cholesky Factorization 67
5.1 Introduction . . . 67
5.2 Tacho: Task Parallel Sparse Factorization . . . 68
5.2.1 2D Sparse Block Matrix . . . 69
5.2.2 Cholesky-by-blocks . . . 71
5.2.3 Implementation Details using Kokkos Task API . . . 75
5.3 Performance Evaluation . . . 77
5.3.1 Task Parallel Cholesky-by-blocks Factorization . . . 77
5.4 Summary . . . 81

6 miniTri: A Data Analytics Mini-Application 83
6.1 Background . . . 83
6.1.1 Linear Algebra Primitives for Graph Algorithms . . . 83
6.1.2 miniTri: A Data Analytics Miniapp . . . 84
6.2 Task-Parallel miniTri Overview . . . 85
6.2.1 Challenges with Linear Algebra Based miniTri . . . 85
6.2.2 Task-Parallel Linear Algebra Based miniTri . . . 86
6.3 Task-Parallel Kokkos/Qthreads miniTri Implementation . . . 87
6.3.1 General Implementation Details . . . 87
6.3.2 Kokkos Task-Parallel Functionality . . . 88

6.3.3 Dynamic Task Dependencies of miniTri 88
6.4 Results 89
6.5 Conclusions/Lessons Learned 91

7 Conclusion **95**

7.1 Goals and Key Results 95
7.2 Continued R&D 96

List of Figures

2.1 Task state transitions for spawn and respawn operations 18

2.2 Tasks execute serial or data parallel 19

2.3 Obligatory example of pure-task based Fibonacci function 20

3.1 Task queue linked lists 30

3.2 Summary atomic linked list push 31

3.3 Summary of almost-atomic linked list pop 31

3.4 Scheduling a task entity 32

3.5 Scheduling a when-all entity 32

3.6 Completing a task or when-all entity 33

3.7 CPU hardware thread team synchronization 35

3.8 GPU warp is divided into thread team members and vector lanes 37

3.9 Team and vector levels of parallel operations 37

3.10 Memory Pool Allocator API 38

3.11 Superblock State 42

4.1 The compressed sparse row graph data structure 50

4.2 A typical MTGL algorithm signature 51

4.3 Calling an MTGL algorithm 51

4.4 basic MTGL includes and namespace 54

4.5 a typical MTGL test program: header information 55

4.6 declaring MTGL graph iterators and property maps 55

4.7 MTGL property map access via generic function 56

4.8 MTGL property map access with Kokkos parallel primitives 56

4.9 applying a functor to an MTGL property map 57

4.10 linear_recurrence functions 57

4.11 vertex and edge reductions with temporary functions 58

4.12 vertex and edge reductions with functors 58

4.13 calling the MTGL breadth_first_search 60

4.14 defining the visitor for MTGL breadth_first_search 60

4.15 finding connected components with the MTGL "shiloach_vishkin" algorithm . 61

4.16 Team-level MTGL library coding 65

5.1 2D Sparse block layout on a matrix (msc10848) selected from the UF collection. 69

5.2 A simplified example of symmetric block nested dissection ordering permuted by Scotch. Left: a given sparse matrix with its natural ordering. Right: a 2D sparse block layout on the reordered matrix. 70

5.3 Cholesky algorithm. The blocks in the 2×2 and 3×3 block matrices that correspond to each other are of the same color. CHOL and TRIU represent Cholesky factorization and the upper triangular part of an input matrix respectively. 71

5.4 Cholesky-by-blocks algorithm on a 2D block layout. 72

5.5 Generated block matrix computations while proceeding on Cholesky-by-blocks. 74

5.6 A task dependence graph for the example illustrated in Fig. 5.5. 75

5.7 Main driver for task spawning from the host execution space. 75

5.8 A task to generate tasks for Cholesky-by-blocks. 76

5.9 Time for level(k) incomplete Cholesky-by-blocks factorization on testbeds: Sandy Bridge, Intel Xeon Phi and IBM POWER8. The time complexity includes both symbolic and numeric factorization. 79

5.10 Tasking overhead for different matrices with different level of fills. 80

6.1 Linear Algebra Based Formulation of miniTri. 85

6.2 Illustrative example of task-parallel approach to linear algebra based miniTri. Arrows represent dependencies between tasks. 87

6.3 Illustration of complex dependences for k-count tasks in miniTri. 89

6.4 Run times (in s) miniTri run times (in s) for Oregon-1 graph for Kokkos/Qthreads implementation for different block size. 90

6.5 Comparison of OpenMP and Kokkos/Qthreads miniTri implementations for Oregon-1 graph (best block size for both methods shown). 91

6.6 Comparison of OpenMP and Kokkos/Qthreads miniTri implementations for email-Enron graph (best block size for both methods shown). 92

6.7 Comparison of miniTri run times (in seconds) for ca-AstroPh and com-Youtube graphs: OpenMP versus Kokkos/Qthreads (best block size for both methods shown). 93

List of Tables

4.1 MTGL Types, Descriptors, Iterators, and Categories 52

4.2 MTGL Breadth-first search visitor API 59

4.3 Basic Kokkos data parallelism through the MTGL interface 61

4.4 Data parallelism and graph iteration 62

4.5 Datasets for MTGL library algorithm experiments 63

4.6 MTGL breadth-first search performance 63

4.7 MTGL PageRank performance 63

4.8 MTGL connected components performance 64

5.1 Test problems selected from the UF sparse matrix collection. 77

5.2 Number of nonzero Cholesky factors resulting from level(k) symbolic factorization; for comparison, the last column shows the number of fill from complete factorization with AMD ordering. 78

6.1 Datasets 90

Chapter 1

Introduction

To sustain scalability on emerging manycore computing architectures (Terascale workstations, Petascale clusters, and Exascale supercomputers) our analysis codes must exploit all opportunities for manycore parallelism. Migrating applications to manycore architectures currently requires scientists and engineers to (1) have detailed knowledge of vendor-specific performance characteristics and constraints, (2) obfuscate essential mathematics in their codes with parallel processing directives, and (3) generate and maintain multiple versions of codes to meet vendor-specific performance requirements. Even vendor-neutral programming models (OpenMP, OpenACC, OpenCL) require architecture-specific knowledge to achieve acceptable performance and pollute mathematical code with parallel processing directives.

Two key R&D projects at Sandia address independent facets of manycore parallelism. Qthreads [1] addresses task parallelism with highly efficient task scheduling algorithms. Kokkos [2] addresses data and vector parallelism through a performance portable interface that minimizes users exposure to architecture-specific details. In this project we integrate these R&D efforts to create a hierarchical task-data parallel capability that is performance portable across manycore architectures. The application programmer interface (API) for this capability is an enhancement of the Kokkos library and the implementation of this capability utilizes an enhancement of the Qthreads library and its task scheduling algorithms.

Our productivity goal is to enable scientists and engineers to program emerging manycore architectures with neither extensive architecture-specific knowledge nor ubiquitous parallel processing directives. This capability should be general and applicable to diverse application domains. We evaluate this *generality* goal by exercising the integrated Kokkos / Qthreads libraries across high performance computing (HPC) domains of consequence to Sandia. For the simulation domain we develop a hierarchical task-data sparse matrix factorization computational kernel used in the solution of sparse linear systems. For the analytics domain we develop a new port of Sandia's Multithreaded Graph Library (MTGL) using the enhanced Kokkos interface.

1.1 Concepts and Terminology

The goal of our R&D is to create new capability for a shared memory directed acyclic graph (DAG) of tasks with internal data parallelism that is easily accessible to application developers (our users). This goal-statement is densely packed with concepts and terminology elaborated as follows.

Task: A *task* is an application's computation that is given to a runtime system for subsequent execution. A task is implemented as a function and parameters with which the function operates. A runtime system can execute independent tasks in parallel on hardware threads of a multicore CPU or manycore GPU.

Dependences and Task DAG: A task may have an "executes after" dependence on one or more other tasks. For example, when task **B** inputs data that is output by task **A** then task **B** must execute after task **A** completes. The collection of tasks and dependences defines a graph where tasks are graph-vertices and dependences are directed graph-edges. This graph cannot contain cycles of dependences, otherwise tasks within the cycle could never be executed. We refer to the parallel execution of a collection of tasks where the inter-task executes-after dependences are enforced as the *task DAG* parallel pattern. In a traditional task DAG parallel pattern each task executes on a single hardware thread and is thus internally serial.

Data Parallel Task on a Thread Team: New manycore architectures have an increasing number of hardware threads where groups of hardware threads share critical hardware resources such as registers and L1 cache. For example, the four hyperthreads of the Intel Xeon Phi architecture and the "warp lanes" of an NVIDIA GPU architecture define a tightly bound group of hardware threads. When these tightly bound hardware threads execute different tasks computing on different data they will compete for shared resources, which typically results in degraded performance compared to one of the hardware threads executing the same tasks sequentially. To effectively use the parallelism offered by tightly bound hardware threads a task should execute on that entire group and utilize the group as a well-coordinated *thread team*. Thread-team coordination is most easily managed and expressed through data parallel operations such as parallel_for, parallel_reduce, or parallel_scan. Thus a task DAG will be most performant on new manycore architectures when tasks run on thread teams with internal data parallelism. The first innovation of our R&D is to support tasks with internal data parallel operations running on thread teams.

Execution Space and Task Policy: A heterogeneous compute node has multiple resources for executing computations; for example, the NUMA regions of multicore CPUs and attached GPUs. Kokkos refers to a resource *where* parallel computations may execute as an *execution space* and the rules or algorithms for *how* to schedule those computations as an

execution policy. Thus, in Kokkos' nomenclature we name a runtime system which manages, schedules, and executes a task DAG as a *task execution policy*, or more succinctly as a *task policy*, on an execution space.

Spawning and Future: *Spawning* is the process of creating a task and inserting it into a task policy for future execution. When a task is spawned a task policy returns a handle for that task, referred to as a *future*. Futures are used to express task dependences and obtain simple return values from completed tasks. For example, an application spawns task **A** and then spawns task **B** with an executes-after dependence on task **A**. When task **B** is created, it may be given a future referencing task **A** so that when task **B** executes it can query the return value from task **A**.

Dynamic Task DAG: A task DAG implementation may require that the complete task DAG be constructed before any task within the task DAG can execute – we refer to this as a *static* task DAG. We require a *dynamic* task DAG where an executing task may spawn new tasks with new dependences. This dynamic task DAG requirement posed significant design and implementation challenges for the GPU execution space, scalability with respect to the number of hardware threads, and scalability with respect to the size of the task DAG. The second innovation of our R&D is to support the dynamic task DAG parallel pattern on a GPU execution space.

Respawn and Non-blocking (non-waiting) Tasks: In a traditional task DAG parallel pattern when a task spawns another task it may wait for that spawned task to complete. Such a wait operation defines an implicit dependence – the waiting task blocks and can only resume executing after the spawned task completes. In order to guarantee that execution of the overall task DAG will make progress a blocked task must relinquish its execution resource so that other tasks can execute on that resource. Considerable runtime mechanisms are required to support blocking and relinquishing execution resources. Supporting blocking and relinquishing resources adds complexity, memory overhead, and runtime overhead to task scheduding and execution. We deemed that this complexity and overhead is unacceptable for the intended HPC algorithms and may not even be feasible to implement for lightweight GPU cores. Instead of a wait operation we provide the *respawn* operation by which an executing task can re-insert itself into the task DAG with new dependences. The respawn operation causes an executing task, upon returning from its function, to be rescheduled for subsequent execution when its new dependences have completed. Thus a traditional operation of task **B** waiting on task **A** can be replaced by task **B** respawning itself with a new dependence on task **A** and then returning. The third innovation of our R&D is the respawn replacement for the traditional wait operations in a dynamic task DAG.

Task Priority: An application's task DAG algorithm may have knowledge of task DAG branches that are in the critical path for completion of the algorithm. An algorithm may

direct the task policy to prioritize execution of tasks in the critical path to more quickly complete the whole algorithm. An algorithm may also prioritize tasks to manage peak memory consumption during task DAG execution.

Memory Pool Allocator: A dynamic task DAG allocates and deallocates memory for tasks and dependences while tasks are executing in parallel. Significant challenges for dynamic memory management are portability to the GPU execution space and scalability with respect to the number of threads concurrently allocating and deallocating memory. Memory allocation and deallocation by the GPU runtime from within a GPU function uses a fixed pool of memory that is allocated on the host before the GPU function can be launched. The fourth innovation of our R&D is a portable and thread scalable memory pool allocator used by the dynamic task DAG runtime on GPUs and other execution spaces.

1.2 Kokkos / Qthreads Integration

Execution policies, thread teams, and data parallelism are existing Kokkos capabilities. Our R&D extends these capabilities for the dynamic task DAG parallel pattern. Scheduling tasks with dependences and executing tasks on a single hardware thread are existing Qthreads capabilities. Our R&D extends these capabilities to support executing a given task on a team of hardware threads. We integrate Qthreads as an additional *back-end* to Kokkos (alongside Kokkos' existing back-ends OpenMP, pthreads, and CUDA back-ends) to enable a Kokkos-based interface for tasks. We also apply Qthreads' task management and scheduling strategies to implement in other Kokkos back-ends the "dynamic task DAG with internally data parallel tasks" pattern.

1.3 Mini-Applications

The developed task DAG capability is evaluated with two mini-applications. The sparse matrix Cholesky factorization mini-application uses a blocked algorithm to define a task DAG of submatrix computations. The triangle enumeration mini-applications defines a task DAG to concurrently find and analyze triangle-cycles within a social network graph.

Chapter 2

Dynamic Task DAG Abstractions, Requirements, and Lessons Learned

A program begins in a single host process thread which typically invokes the program's `main` function. In an HPC application this host process thread typically calls a message passing interface (MPI) library's initialize function and obtains an MPI communicator for subsequent distributed memory parallel operations. The Kokkos and Qthreads libraries are solely concerned with shared memory parallel operations, and as such are decoupled from distributed memory parallel operations.

A host thread process initializes Kokkos, which in turn initializes one or more back-ends that execute parallel computations within execution spaces. One of these back-ends uses Qthreads to execute parallel computations on CPU and Intel Xeon Phi architectures. Initialization acquires execution resources (*e.g.*, hyperthreads, attached GPU) on which subsequent intra-node parallel operations execute. These include simple data parallel operations (*e.g.*, parallel_for over a range $[i, j)$), hierarchical data parallel operations using thread teams, and hierarchical task-data parallel operations.

2.1 Life-cycle of a Task in a Dynamic Task DAG

The life-cycle of a task in a dynamic task DAG is presented from an abstract perspective with diagrams and specifications, and then again from an illustrative code perspective.

2.1.1 An Abstract Perspective

A user's task is implemented by a *closure* with the states and life cycle illustrated in Figures 2.1 and 2.2. A closure is either a C++ *functor* which is a C++ class that has a member function with specified calling arguments, or C++11 *lambda* expression which is an anonymous functor that is automatically generated by the C++ compiler. The observable states of a task are constructing, waiting, executing, and complete.

A spawn operation allocates memory, constructs, and inserts a task into a task policy's

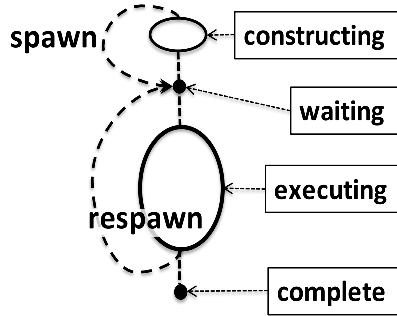


Figure 2.1: Task state transitions for spawn and respawn operations. When a user spawns a task it is allocated, constructed, and insert into a task policy's collection of waiting tasks. When a user respawns an existing task it is re-inserted into a task policy's collection of waiting tasks.

collection of waiting tasks. The spawn operation calls a functor's class constructor or invokes the lambda's capture mechanism within the thread that performs the spawn operation. While the class constructor or lambda capture is executing the task is in the constructing state After the construction finishes a spawn operation inserts the task into the task policy's collection of waiting tasks, and the task is in the waiting state. A task may be spawned with one or more execute-after dependences. A task is in the waiting state is either waiting on its dependences to complete or is ready to execute and waiting for the task policy to select the task for execution.

A task policy selects a task A that is waiting and ready to execute. The task's execution function, the C++ `operator()` member function with the required calling arguments, is called in the designated execution space. While this function is executing the task is in the executing state. If this function returns without having requested to be respawned the task policy completes task A by (1) updating any tasks waiting on task A and (2) destroying and deallocating task A if no futures or dependences exist that reference task A. A task that is not deallocated but has completed execution is in the complete state.

During its execution task A may request to be respawned with or without new dependences. In this case, when task A's execution function returns the task policy will re-insert task A into the collection of waiting tasks.

A task policy executes a serial task on a single thread and executes a data parallel task on a thread team (Figure 2.2). When a data parallel task executes its execution function is called by each member of the thread team and the task's C++ class instance or lambda-capture is shared by all threads of the thread team. These threads are guaranteed to execute concurrently. Thus a data parallel task's execution function must be carefully implemented to avoid concurrent access race conditions on the shared task data. For example, only one member of the task's thread team should request a respawn for the task.

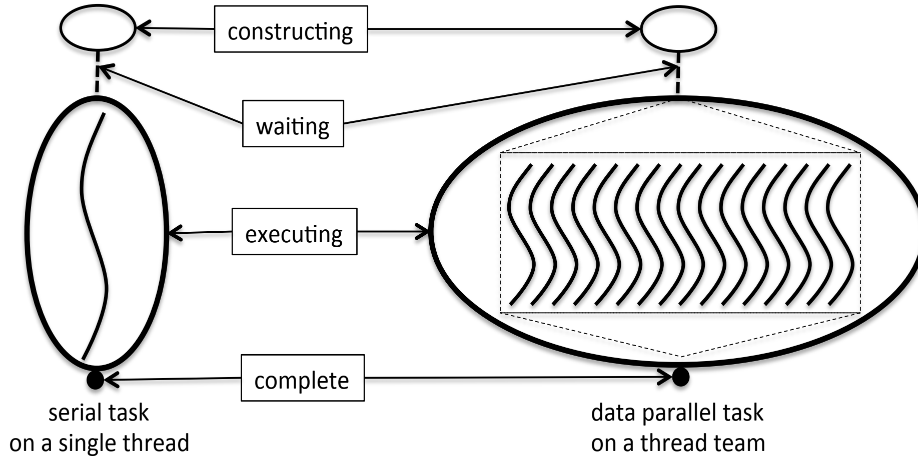


Figure 2.2: Tasks execute serial or data parallel. A task policy selects a task that is ready to execute from the waiting collection and executes the task on a single thread or thread team. If the task does not respawn when the task finishes executing it is complete.

2.1.2 An Illustrative Code Perspective

We repeat the overview of a task states and lifecycle, this time elaborating from an illustrative code perspective. For this illustration we use the obligatory Fibonacci function with the naive, purely task based implementation presented in Figure 2.3. While this example is an extremely inefficient implementation of the Fibonacci function, it provides an appropriately simple code-based illustration of our task DAG abstractions and mechanisms.

The `fibonacci` function first constructs a `TaskPolicy` object (Fig. 2.3 line #6). A task policy executes tasks in an execution space that is specified by the task policy's `ExecSpace` template argument. A task policy supports dynamic memory allocation and deallocation for task memory through a thread scalable memory pool within a memory space. The memory space and size of this memory pool are arguments to the task policy constructor.

The host process calls the `host_spawn` function to spawn the initial $f(N)$ task and obtain a `Future` which references the spawned task (Fig. 2.3 line #7). The spawn function is given a `FibonacciTask` functor and the `TaskSingle` parameter indicating a single thread task. Spawning a task from the host process has different implementation considerations than spawning a task from within an executing task. For example, the mechanism for capturing the task's execution function is different when spawning from the host process or within a GPU task. The host process then waits for *all* ready tasks contained within in the task policy to complete (Figure 2.3 line #8). The task policy may not execute any tasks until the host process calls the `wait` function (Fig. 2.3 line #8). For example, when using a GPU execution space the `wait` function launches a GPU kernel to execute the task DAG. Once the all ready tasks in the task policy have completed the `wait` function returns and the $f(N)$ task's future is queried for the resulting value (Fig. 2.3 line #9).

```
template< typename ExecSpace >
long fibonacci( long n )
{
  using memory_space = typename ExecSpace::memory_space ;
  const size_t memory_pool_size = /* ... */ ;
  TaskPolicy< ExecSpace > P( memory_space() , memory_pool_size );
  Future< ExecSpace , long > F = P.host_spawn( FibonacciTask(P,n), TaskSingle );
  wait( P );
  return F.get();
}

template< typename ExecSpace >
struct FibonacciTask {

  using value_type  = long ;
  using policy_type = TaskPolicy<ExecSpace> ;
  using future_type = Future<ExecSpace,value_type> ;
  using member_type = typename policy_type::member_type ;

  policy_type P ;
  future_type child[2] ;
  value_type  N ;

  void operator()( const member_type & , value_type & result )
  {
    if ( N < 2 ) { result = N ; }
    else if ( child[0].is_null() ) {
      child[0] = P.task_spawn( FibonacciTask(P,N-1), TaskSingle );
      child[1] = P.task_spawn( FibonacciTask(P,N-2), TaskSingle , TaskHighPriority );
      P.respawn( this , P.when_all(2,child), TaskLowPriority );
    }
    else {
      result = child[0].get() + child[1].get();
    }
  }

  FibonacciTask( const policy_type & argP , long argN )
    : P(argP), child{}, N(argN) {}
};
```

Figure 2.3: Obligatory example of a simplistic, pure-task based Fibonacci function, $f(N) = f(N-1) + f(N-2)$, implemented with Kokkos' dynamic task DAG functionality.

A task's resulting value is declared by the functor's `value_type` (Fig. 2.3 line #15). The functor's `operator()` implements the task's execution function. This function is called with a task policy member and a value object in which the function outputs its result. The task policy member argument defines the context in which the task executes; for example, which member of a thread team. In the Fibonnacci illustration all tasks are single thread so the task policy member argument can be ignored.

In the Fibonacci illustration a functor's `child` futures are initially *null* – they do not reference a task. The first time a Fibonacci task is called with $2 < N$ wo child Fibonacci tasks ($f(N-1)$ and $f(N-2)$) are spawned and their futures are saved in the `child` variable (Fig. 2.3 lines #28-29). Since these spawn operations occur within a running task the `task_spawn` function is used instead of `host_spawn`. The $f(N)$ task cannot explicitly wait on the two spawned tasks so the $f(N)$ task implicitly waits by requesting respawning with dependences on the `child` tasks (Fig. 2.3 line #30). A respawn operation declares new dependences, may declare a different priority, but cannot change the task' original single

thread / thread team designation. Upon returning from its execute function the $f(N)$ task is re-inserted into the task policy's collecton of waiting tasks.

A task may be initially spawned with D_0 dependences and subsequently respawn itself K times, each time with D_k dependences. Supporting a dynamic number of dependences adds complexity and potentially degrades performance in a task DAG implementation. To maintain a simple and performant implementation a task is spawned or respawned with at most one `Future`. Multiple execute-after dependences are supported by creating a `when_all` aggregate future (Fig. 2.3 line #30). The `when_all` function returns an future that references an aggregate collection of tasks as opposed to a single task. This future is complete when all tasks in the aggregte collection are complete.

In the Fibonacci illustration a $f(N)$ task request respawning with new dependences on $f(N-1)$ and $f(N-2)$ `child` tasks. The $f(N)$ task is called again after the child tasks complete and their results can be queried. The second call to a $f(N)$ task can query $f(N-1)$ and $f(N-2)$ result to compute its own result (Fig. 2.3 line #33). This time the task does not respawn will be complete when the execution function returns.

Selecting ready tasks for execution: Spawning or respawning a task optionally declares the task to have high, regular, or low priority; where regular priority is the default. Most of Kokkos' back-end task policies choose tasks from those that are ready to run as follows. First, when multiple tasks are ready-to-run tasks with higher priority are chosen over lower priority. Second, when priorities are equal then a last-ready-first-run strategy is used; *a.k.a.*, a last-in-first-out (LIFO) strategy.

2.2 Lessons Learned and Evolving Abstractions

Research and development of the dynamic task DAG capability transpired through several prototyping iterations. Each prototype was unit tested and evaluated by using the prototype mini-applications. Evaluation was initially performed on CPU multicore architectures, then Intel Xeon Phi manycore architectures, and finally NVIDIA GPU architectures. Each iteration resulted in lessons learned regarding usability, performance, and hardware constraints; leading to design improvements and re-implementation.

2.2.1 Spawning and Dependences

When spawning or respawning a task the number of dependences is not determined until at runtime. Our first prototype allocated task data when a task is spawned to accomodate a specified number of dependences. However, when initially spawned the number of dependences required for a subsequent respawn request may not be known. This became problematic because an algorithm must guess at an upper bound for respawn dependences

– a usability impediment. Our lesson learned is to separate task allocation from multiple-dependence allocation. This is accomplished by allowing a task to have at most one direct dependence, which can be an independently allocated aggregate *when all* dependence. This allows an application to spawn and respawn tasks without concern for the number of dependences that a task may be given at some future respawn step.

2.2.2 Compatibility of Serial and Thread Team Tasks

A task executing on a thread team requires a mechanism for coordinating the thread team; *e.g.*, partitioning data parallel work among the threads of the team. In contrast, a task executing on a single thread does not have this requirement. Because of these different requirements our initial prototype design defined two different task execution function interfaces for single thread tasks and thread team tasks. This initial design was problematic for algorithms that require the flexibility to choose whether a given task was to execute on a single thread or thread team.

In our final design we define the same interface for both single thread and thread team tasks. In this interface, shown in Figure 2.3 on line #24, an execution function is always called with a thread team `member_type` argument. If a task is guaranteed to be called on a single thread this argument can be ignored, as in the Fibonacci illustration. Otherwise the `member_type` argument is used to coordinate thread team parallel operations.

2.2.3 Memory Management – Thread Scalability

Our first prototype on a multicore CPU architecture used the runtime's standard memory allocation and deallocation functions. With the CPU's small number of hardware thread scalability was not a concern. When progressing to the Intel Xeon Phi manycore architecture these standard runtime functions became a performance problem with frequent concurrent allocation and deallocations. To call these standard allocation and deallocation functions on a GPU requires pre-allocation of a pool of memory from the host process, and this pre-allocation cannot be subsequently released. This is problematic because when a task DAG algorithm completes all of its allocated task memory is no longer needed but that memory cannot be released. Thus we developed a portable and thread scalable memory pool allocator through which a task DAG memory pool is allocated when the task policy is constructed and deallocated when the task policy is destructed. Details of the portable, thread scalable memory pool allocator are given in Section 3.4.

2.2.4 Memory Management – Task Scalability

Integrating a memory pool into the task policy posed new challenges for our mini-applications. In the first CPU-only prototype algorithms that had access to practically unlimited memory

of the large CPU node, now these algoirthms had to address a fixed amount of memory for the task DAG. The first algorithmic observation is that even when the entire task DAG is huge only a small fraction of tasks are running at a given moment. An algorithm need only have enough ready tasks in the dynamic task DAG to keep all hardware threads busy executing tasks, and ensure that as executing tasks complete there are ready tasks to begin executing. Beyond this number waiting or completed tasks merely consume memory without contributing to the progress of the algorithm. Thus for scalability with respect to the end-to-end size of a task DAG an algorithm must now also consider the "high water mark" of memory allocated for tasks. This high water mark is governed by the rate at which tasks are spawned (allocated) versus the rate at which tasks are completed and then deallocated.

A task DAG algorithm can manage the rate of task spawning versus the rate of task completion by spawning a *driver* task that, when it executes, iteratively spawns the algorithm's large number of *computational* tasks. When computational tasks are spawned within an iteration the spawning rate can be "throttled" by periodically exiting the iteration, respawning the driver task at lower priority than the computational tasks, and then resuming iterative task spawning when the driver task is called again. Respawning the driver task at a lower priority guarantees that execution of the driver task will only resume when there are no higher priority computational tasks ready to execute. As long as computational tasks are deleted upon completion an algorithm can managed the task high water mark.

2.2.5 Memory Management – Task Lifetime

A *static* task policy allocates all tasks and dependences before any task begins executing and does not modify or deallocate a task until all tasks have completed. In contrast a *dynamic* task policy allocates, modifies, and deallocates tasks and their dependences while other tasks are executing. In a dynamic task policy the lifetime of a task begins when a spawn operation allocates the task and ends when the task is complete and no `Future` exists that references that task, and the task can be deallocated. One or more futures may reference a task; therefore, these futures have shared ownership of the task along with the task policy; *i.e.*, the task's memory must be deallocated when the last `TaskPolicy` or `Future` reference is destroyed.

The closure of a task may contain futures that reference other tasks. In the Fibonacci example (Figure 2.3) the $f(N)$ task holds futures for the $f(N-1)$ and $f(N-2)$ These futures, and the entire closure, are unused and unnecessary once the task completes. Furthermore, the existence of these futures prevents their associated tasks from being deallocated until the closure's C++ destructor is applied to destroy the closure. Therefore once a task is complete it associated closure is immediately destroyed even though the task's memory may not be deallocated. In the Fibonacci example this means that the C++ `FibonacciTask` destructors are applied to the $f(N-1)$ and $f(N-2)$ closures before the respawned $f(N)$ task is executed for the second time; which allows the "child" tasks of $f(N-1)$ and $f(N-2)$ to be deallocated as well.

2.2.6 Critical Paths and Short Paths

A task DAG algorithm may have knowledge of task DAG branches that are in a critical path for completion of the algorithm or are in a *short* path, which is likely to more quickly complete if executed. With this knowledge an algorithm can prioritize execution of tasks in the critical path to complete the algorithm more quickly, or prioritize tasks in short paths so that those tasks can be deallocated. In the Fibonacci example the $f(N-2)$ task is spawned with higher priority than the $f(N-1)$ task because it has a shorter path to completion and can then deallocate tasks on that branch.

2.2.7 CUDA Porting

Porting to NVIDIA CUDA was the high risk "stretch goal" for our project. We anticipated the non-blocking (non-waiting) challenge and conceived the respawn strategy to portably address this constraint. We anticapated the task memory management challenge and developed the memory pool to portability address this requirement. We anticipated that separate compilation of GPU task execution functions requires the use of CUDA *relocatable device code* and managing the resulting GPU-versus-CPU function pointers. However, we did not anticipate that implementing a task policy for CUDA would push NVIDIA's nvcc compiler and linker technology to the "breaking point."

Our early simple unit tests of required CUDA features were successful using nvcc v7.5, the most recent release of nvcc. However, when the prototype task policy was ready to build the nvcc v7.5 linker was unable to link the prototype, failing with an internal error. We leveraged our collaborative relationship with NVIDIA to quickly diagnose the error – discovering that it was a known bug which would be fixed in the yet unreleased nvcc v8.0. We then heavily leveraged our long-term collabortive relationship with NVIDIA to obtain early access to a pre-release nvcc v8.0. With this version of nvcc we successfully ported to CUDA and met our high-risk "stretch goal."

> We appreciate the above-and-beyond-expectations support we received from NVIDIA. Our CUDA porting stretch goal could not have been accomplished without this support.

2.2.8 GPU Thread Team

Our prorotype task policy for CUDA mapped thread team tasks onto CUDA thread blocks. NVIDIA GPU hardware is designed around a performance point of thread block sizes between 128 and 512 "hardware threads," with 1 and 1024 as absolute lower and upper bounds. These hardware threads are grouped into 32-thread *warps* that effectively execute lock-step on a single instruction stream. Thus when a computation selects a block size of N the computation is assigned enough 32-thread warps to provide N threads.

We learned through our mini-applications that algorithms using hierarchical task-data parallelism with large task counts are likely to use only modest amounts of data parallelism within tasks. Thus the mapping of thread team tasks onto entire CUDA thread blocks left many GPU threads idle. As such we redesigned the CUDA task policy back-end to map thread team tasks onto warps instead of blocks.

2.3 Final Peer Review – Nomenclature Revisions

A final Sandia-internal peer review of the Kokkos task DAG capability and API was held on September 8, 2016. This peer review generated recommendations to revise API nomenclature and enhance functionality as follows.

In this document we refer to the component which manages the execution and lifetime of an entire task DAG as the *task policy*. The peer review recommendation is to change the name of this component to *task scheduler*, and apply the term *task policy* to how a particular task is to be executed. This nomenclature is consistent with task DAG capabilities such as in Uintah [3] and in Charm [4]. This nomenclature is also consistent with the Kokkos data parallel API where execution policies are applied when dispatching data parallel kernels.

Task thread teams are currently a fixed size that is deduced from the underlying hardware (Section 3.3.2). A requested enhancement is to allow the thread team size to be selected by the application.

Chapter 3

Task Schedulers and Runtime Systems

Task parallelism delegates responsibility for scheduling parallel computations from the application programmer to a task policy and it underlying runtime system. Design and implementation of a runtime system can have a profound effect on the efficiency and scalability of the application. An advantage of applications using Kokkos' task policies is that many different back-end implementations are available without changing the application source code. We developed several implementations described in this chapter.

3.1 Qthreads Back-end

The Qthreads multithreading library is a C-based API and run time system inspired by the Tera/Cray MTA/XMT architectures. Analogous to those hardware barrel-processor machines, it is designed to provide software support for many user-level tasks in flight with fast synchronization using full-empty bits (FEBs). Qthreads schedules its user-level tasks, called qthreads, onto long-lived worker pthreads that are pinned to cores in topology-aware configurations. Several different task scheduling strategies and locality-based optimizations have been developed for Qthreads, making it an attractive choice for a performant Kokkos tasking back-end.

Adapting Qthreads for use with Kokkos required the addition of some new functionality while leveraging key existing features as well. Qthreads provides performant task creation, scheduling, execution, and synchronization. Dependence resolution uses the "preconditioned task" feature, in which completions of dependences trigger the movement of tasks into ready queues in an event-driven manner. New support was needed for the "task team" concept, in which a coordinated group of tasks execute synchronously on tightly-coupled hardware resources, such as a small group of cores sharing a cache or a set of hardware threads on the same core. Qthreads scheduling had in the past been centerd on a model of singleton tasks load balanced among distributed queues by work stealing without regard for coordinating tasks (other than ensuring that dependences are fulfilled). While one solution to add the needed functionality would have been to create a new scheduler, the chosen solution was to modify the existing scheduler. Two new mechanisms were added: cloned tasks and local priority queues, described below. Note that they were incorporated into Qthreads in such a way as to be useful not only for Kokkos but also for other libraries and applications using

Qthreads.

Cloned Tasks The new task cloning facility in Qthreads allows a task to be spawned with a `clone_count` parameter indicating the number of clones, i.e., identical instances, of that task to be executed. Each clone is executed independently. To implement data-parallelism in tasks, Kokkos creates a set of clones for each task corresponding to the number of threads in the team. Since the clones are executed independently, the body of each task includes synchronization among the tasks. Although cloned tasks are identical, Kokkos queries the thread identifier for the thread that is executing each task and uses that number as an index into the work (iteration space) and data (views).

Local Priority Queues The other major Qthreads capability added to support Kokkos tasking is the local priority queue. The default scheduler in Qthreads divides the available cores into locality domains, known as shepherds. Each domain has its own queue and a set of nearby threads that draw from the queue. Work stealing between the queues of the different shepherds maintains load balance even in the presence of irregular and unpredictable workloads. (Work stealing is a well-known process by which idle worker threads draw work from remote queues where more ready tasks are available.) For data-parallel Kokkos tasks, all related tasks, implemented as cloned tasks as described above, should be executed by a team of nearby threads. Thus in Qthreads such tasks should be queued and executed on the same shepherd's queue and not stolen by other shepherds' worker threads. Furthermore, each of the related (cloned) tasks should be executed simultaneously within the team of threads, each on a different worker.

In addition the default queue on each shepherd, we have added a new local priority queue. "Local" indicates that each queue is particular to a shepherd. "Priority" indicates that an idle thread within the shepherd first checks this queue for available tasks to execute before checking the shepherd's default queue. By enqueueing together the set of cloned Qthreads tasks corresponding to a data-parallel Kokkos task, the runtime ensures that all clones will remain on that shepherd and dequeued successively by each worker in the shepherd until all are executing, without dequeing any intervening unrelated tasks from the shepherd's default queue.

Distributed Queues One of the challenges for a task runtime is efficiently managing the increasing number of cores per NUMA domain. Having one queue in memory that all cores share creates a significant contention bottleneck. To address this issue, we've implemented a new scheduler for Qthreads that has multiple queues per NUMA domain that are shared between workers. Work is distributed across these queues uniformly, preventing a single point of contention. This approach has significant benefits for data-parallel tasks, as the sequential bottleneck of each worker dequeueing a task clone from a shared queue for a data-parallel task does not scale to large core counts. Building on the new scheduler, we've started implementing an interface for data-parallel tasks that scales better than the `clone_count`

interface outlined above, to be completed with follow-on funding after the LDRD ends.

3.2 Prototype Kokkos Serial, Pthreads, and CUDA implementations

We developed prototypes for Kokkos' Serial (non-parallel) and Threads (using raw pthreads) to facilitate rapid evaluation of the task policy interface and design. We also developed mini-applications using these prototypes to evaluate the portability, performance, and usability of the task DAG functionality. Numerous lessons were learned and our abstractions evolved accordingly as described in Section 2.2. These lessons learned also led to significant improvements in the software design and implementation of the second generation task DAG capability developed in the final months of this LDRD.

The early Threads prototype provided an R&D framework for exploring lock free task DAG scheduling algorithms. This prototype was then ported to the Cuda back-end where race conditions were exposed, analyzed, and addressed. This initial lock free implementation enabled an in-depth analysis of the task life-cycle states, state transitions, and how to implement lock free transitions.

At the core of the prototype each task had an explicit state variable that reflected the conceptual states of a task (Figures 2.1 and 2.2). A task's state was explicitly expressed and changes to its state variable were applied atomically. We observed that the task's internal data structure had a one-for-one derived state that was also updated atomically. Thus the explicit state and the internal data structure state were redundant and maintaining both resulted in redundant atomic updates to data.

Each prototype back-end had a fully independent implementation which resulted in some amount of redundantly implemented functionality and duplicated effort for testing that functionality. This motivated a new design of the final back-ends that shares implementation wherever feasible.

3.3 Final Kokkos Serial, OpenMP, and CUDA implementations

The final implementions are based upon Kokkos' Serial, OpenMP, and CUDA back-ends; the OpenMP back-end replaced the Threads (raw pthreads) back-end at the request of early-adopter application developers. These implementations share a common task DAG queue data structure and foundational operations for managing this data structure. However, due to significant differences in the underlying runtimes these implementations cannot share mechanisms for executing a task DAG.

3.3.1 Managing Task DAG Queues

A task policy owns a collection of spawned tasks that are organized into multiple queues. Each queue is a simple uni-directional linked list of spawned tasks that can be efficiently updated through atomic operations. Thus each queue can be simply defined as the head of a linked list where each task is the member of a linked list, as illustrated in Figure 3.1.

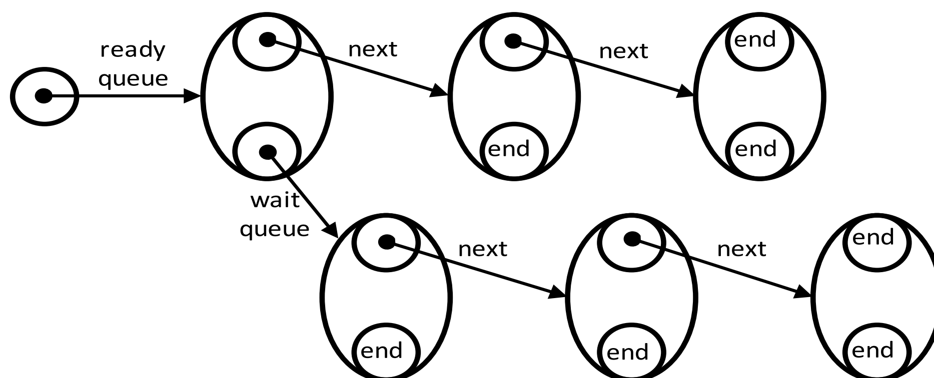


Figure 3.1: Each *ready queue* is defined by the head of a linked list of tasks. Each task is a member of exactly one linked list, and is also the head of a *wait queue* of tasks and when-all entities waiting on that task. The end of each linked list is denoted by an *end* value.

A *ready* queue holds tasks that are ready to execute – tasks that do not have an incomplete dependence. A ready queue is defined for each priority and whether tasks in the queue are to execute on a single thread or a thread team. In Figure 3.1 a ready queue is denoted by the head of a linked list in upper left corner, and the linked list itself is denoted by the linkages labeled *next*. Thus given high, regular, and low priorities there are six ready queues each implemented by the head of a linked list.

A *wait* queue holds tasks and *when-all* entities that are waiting for execute-after dependence(s) to complete. Each task or when-all entity holds the head of a linked list of tasks (or when-all entities) waiting for that task to complete. In Figure 3.1 a wait queue is denoted by the head of a linked list in the bottom of each task or when-all entity, and the linked list itself is denoted by the linkages labeled *next*.

A given task or when-all entity will reside in at most one ready or wait queue. An *end* value is used to denote the termination of a linked list queue. Thus when the head of a ready or wait queue has the *end* value that queue is empty.

Push and Pop Operations

A linked list queue is updated with thread safe atomic push operation and almost-atomic pop operations. The push operation is applied to both ready and wait queues. The pop operation is only applied to ready queues. These operations, as summarized in Figures 3.2 and 3.3, and are shared by Kokkos' Serial, OpenMP, and CUDA implementations.

```
1  // Push 'x' to the head of the linked list
2  bool push( entity ** head , entity * x )
3  {
4    entity * y = *head ; // Query current head
5    while ( lock != y ) {
6      x->next = y ; // memory fenced assignment
7      entity * z = y ;
8      // Atomically attempt to replace the head of the queue,
9      // failure returns the current head.
10     y = atomic_compare_exchange(head,y,x);
11     if ( z == y ) return true ; // success
12   }
13   x->next = 0 ;  // memory fenced assignment
14   return false ; // queue was locked
15 }
```

Figure 3.2: Summary of atomic push operation for updating linked list queues. The push operation pushes a new entity to the head of the linked list and the previous head becomes the next entity.

```
1  // Pop head from the linked list
2  entity * pop( entity ** head )
3  {
4    entity * y = *head ;       // Query current head
5    while ( end != y ) {       // If queue is not empty
6      if ( lock == y ) y = 0 ; // Do not attempt to re-lock a locked queue
7      entity * x = y ;
8      // Atomically attempt to replace head of list with a lock value,
9      // failure returns the current head.
10     y = atomic_compare_exchange(head,y,lock);
11     if ( x == y ) {  // this pop locked the queue
12       // Atomically unlock the queue by setting head to next entry.
13       atomic_exchange( head , y->next );
14       break ; // success
15     }
16   }
17   return y ;
18 }
```

Figure 3.3: Summary of almost-atomic pop operation for updating linked list queues. The pop operation pops the head of the linked list and the list's next entity becomes the new head.

We say the pop operation is *almost* atomic because there is a brief number of execution steps in which the head of the queue has a *lock* value. On line #10 in Figure 3.3 an attempt is made to lock the queue by atomically setting a *lock* value. If this lock succeeds then the next entry in the linked list is set to the head of the queue (Fig. 3.3 line #13) and the former head entity has been successfully popped. The lock fails when either another concurrent pop operation won the "race" to pop the head entity, or a concurent push operation won the "race" to push a new head entity.

Scheduling

A task or when-all entity is scheduled by pushing it into the appropriate ready or wait queue as summarized in Figures 3.4 and 3.5. A task has at most one execute-after dependence so

the scheduling operation (Fig 3.4 pushes the task into the dependence's wait queue, or if there is no dependence or the one dependence is complete pushes the task into the appropriate ready queue. A when-all entity has many dependences so the scheduling operation (Fig 3.5 attempts to push the entity into one of the incomplete dependences' wait queue. If all of the when-all entity's dependences are complete then the when-all entity transitions from waiting to complete.

```
1  void schedule_task( entity * x , entity * dep )
2  {
3    // Task is ready to execute if there is no execute-after dependence
4    // or the execute-after dependence's wait queue is locked.
5    bool is_ready = ( 0 == dep ) || ( ! push_task( & dep->wait , x );
6
7    if ( is_ready ) {
8      entity ** head = ready_queue( x ); // ready queue requested by this task
9
10     while ( ! push( head , x ) ); // Push to head of ready queue
11   }
12 }
```

Figure 3.4: A task entity is scheduled by pushing it into the execute-after dependence's wait queue or the appropriate ready queue.

```
1  void schedule_when_all( entity * x )
2  {
3    bool is_complete = true ;
4
5    for ( int i = x->dep_count ; 0 < i && is_complete ; ) {
6
7      // Swap dependence for zero
8      entity * dep = atomic_exchange( x->dep_array + i , 0 );
9
10     if ( y != 0 ) { // If non-zero attempt to push to wait queue
11       // When 'dep' is complete its wait queue is locked and push fails
12       is_complete = ! push( & dep->wait , x );
13     }
14   }
15
16   if ( is_complete ) { // all dependences are complete
17     complete( x ); // this when-all is complete
18   }
19 }
```

Figure 3.5: A when-all entity is scheduled discovering which dependence is not complete and pushing then entity into this dependence's wait queue. If an incomplete dependence is not discovered then the when-all dependence is complete.

When a task entity completes executing, or when all dependences of a when-all entity become complete, then the entity's waiting entities must be removed from the wait queue and re-scheduled as summarized in Figure 3.6. This state transition is performed on exactly one thread that has exclusive access to the entity's data wait queue. Thus removing entities from a wait queue does not need to apply the almost-atomic pop operation.

When task or when-all entity's wait queue is locked (Fig 3.6 line #5) then that entity is observably in the complete state. When an entity is complete then new execute-after dependences are not allowed to be added to that entity. Thus the push operation will fail for that entity's locked wait queue as per line #5 in Figure 3.2.

```
void complete( entity * x )
{
  // Atomically obtain the head of the wait queue and
  // lock the wait queue against subsequent push operations.
  entity * y = atomic_exchange( & x->wait , lock );

  if ( y != lock ) { // Succeeded in locking the wait queue
    // This 'x' entity is complete so run its wait queue
    // and re-schedule each entity that had an execute-after dependence.
    while ( y != end ) {
      entity * next = atomic_exchange( & y->next , 0 );
      schedule( y );
      y = next ;
    }
  }
}
```

Figure 3.6: When a task or when-all entity becomes complete all waiting entities are rescheduled.

Spawn, respawn, and creating a when-all

A spawn operation accepts a user's closure and a small set of scheduling parameters (Fig. 2.3 lines #7 and 28-29). Memory is allocated for the task from a memory pool (Section 3.4), the user's input closure is copied into that memory, task scheduling parameters are set, and the task is scheduled into a queue (Fig. 3.4).

A respawn operation is called by a task's execute function and accepts the executing task's `this` pointer and a small set of rescheduling parameters (Fig. 2.3 line #30). The task's scheduling parameters are updated and when the task's execute function returns the task is rescheduled into a queue (Fig. 3.4).

The create when-all operation accepts a set of execute-after dependences (Fig. 2.3 line #30). Memory is allocated for the when-all entity, input dependences are copied into this memory, and the when-all entity is scheduled into a queue (Fig. 3.5).

3.3.2 Executing a Task DAG

The task DAG queue data structure and queue-management operations are shared by Kokkos' Serial, OpenMP, and CUDA implementations. However, the mechanisms for executing tasks on these back-ends are signficantly different. These differences are divided into *driver* mechnanisms and *thread team* mechanisms.

CPU Serial "thread team" and driver

Kokkos' Serial "thread team" implementation is trivial as the team size is fixed at one. The task DAG execution driver is almost as trivial; ready queues are iterated in priority order, tasks are popped from a ready queue their execute functions are called, and the driver loops

until all ready queues are empty.

Since there is only one thread executing the driver function new tasks could be spawned without an opportunity for existing ready tasks to execute. Such a scenario can lead to the task DAG having the maximum memory high-water-mark for allocated tasks. To mitigate this scenario the Serial implemention's spawn operation will execute any ready tasks before allocating memory for the to-be-spawned task. When these tasks complete and their futures are destroyed memory allocated for tasks is deallocated, freeing space for the spawned task to be allocated and reducing the potential memory high-water-mark.

CPU OpenMP thread teams

On multicore CPU and manycore Intel Xeon Phi architectures tasks may execute on thread teams. We assume that a task DAG algorithm will have a large number of potentially concurrent tasks where each task has a modest amount of internal data parallelism (recall Section 2.2.8). On CPU-like architectures each task is executed on a core and any hyper-threads on that core define a thread team for the task. Thus an executing task has exclusive access to per-core CPU resources such as L1 data and instruction caches.

CPU OpenMP thread team barrier

Barrier synchronization of a thread team is a performance critical operation. Team barriers are used to synchronize thread teams' use of shared memory, perform thread team collective operations such as parallel reduce and scan operations, and within the task DAG execution driver. We use the barrier algorithm summarized in Figure 3.7 which minimizes the number of shared variables accessed and minimizes writes to those shared variables.

The variable `sync_shared` is a cache-aligned array of two 64bit integer values that is shared by the team. Within each call to `team_barrier` each thread team member writes exactly one byte within this array to a designated value, and then spin waits for all team members to write their respective bytes. The synchronization variable alternates between calls to `team_barrier` and the synchronization value is flipped every other call.

CPU OpenMP driver

Kokkos' OpenMP task DAG execution driver runs whenever a user calls the `wait` function on the task policy. The driver opens a simple OpenMP parallel region (`#pragma omp parallel`) and executes a worker function on each OpenMP thread in that region. Threads, and thus worker functions, are grouped into disjoint thread teams. The *team lead* thread within each thread team iterates the ready queues in priority order attempting to pop a task a ready queue (Fig. 3.3).

```
void team_barrier_init() {
  sync_shared[0] = 0 ;
  sync_shared[1] = 0 ;
  for ( int i = 0 ; i < team_size ; ++i ) {
    sync_value |= int64_t(1) << (8*i);
    sync_mask  |= int64_t(3) << (8*i);
  }
}
void team_barrier() {
  // alternate between two team-shared synchronization variables:
  int64_t volatile * const sync = sync_shared + ( sync_step & 0x01 );

  // this thread's portion of team-shared synchronization variable:
  int8_t volatile * const sync_self = ((int8_t volatile*) sync) + team_rank ;

  // Write a single byte to signal the team that this thread has arrived.
  *sync_self = int8_t( sync_value & 0x03 );

  // Spin-wait for whole team to arrive at this statement.
  while( sync_value != *sync );

  // An alternating pattern for the synchronization variable and value
  // minimizes writes to the synchronization variable and prevents confusion
  // between iterative calls to the barrier. If team_size == 4 then:
  //   iteration 0:  while( sync_shared[0] != 0x0000000001010101 );
  //   iteration 1:  while( sync_shared[1] != 0x0000000001010101 );
  //   iteration 2:  while( sync_shared[0] != 0x0000000002020202 );
  //   iteration 3:  while( sync_shared[1] != 0x0000000002020202 );
  //   iteration 4:  while( sync_shared[0] != 0x0000000001010101 );
  //   etc.

  ++sync_step ; // how many times the barrier has been called

  if ( 0 == ( 0x01 & sync_step ) ) { // every other step
    sync_value ^= sync_mask ;        // flip the sync value
  }
}
```

Figure 3.7: On CPU architectures the hardware thread team of up to eight threads synchronizes by each thread setting a byte within a shared synchronization variable to a specific value and then all threads wait for all team members' values to be set.

The team lead broadcasts the popped task's pointer all team member threads, or *end* token if the ready queues were empty. If the task is to be run on a thread team then all team member threads call the task's execute function; otherwise only the team lead calls the task and all other team member threads remain idle. When the task's execute function returns the team lead either reschedules the task if it respawned or transitions the task from the executing state to the complete state. The team then loops back to pop another task from a ready queue.

All threads in the OpenMP back-end must exit the worker loop together. This condition requires synchronous, global knowledge of whether all ready queues are empty *and* whether a task is currently executing and could spawn new tasks. To satisfy this requirement the OpenMP driver maintains a shared count of how many tasks are currently executing *and* how many tasks are ready to execute. This count is atomically incremented whenever a task is pushed into a ready queue and atomically decremented after a running task completes and its waiting tasks are rescheduled. When the "executing plus ready" task count is zero the

worker loop is exited the entire OpenMP parallel region.

If the number of OpenMP threads in the parallel region is exactly one then task DAG execution has the same memory high-water-mark concern as the Serial implementation. In this situation the mitigation strategy is the same, whenever a task is spawned within a one-thread OpenMP driver all ready tasks are executed before the to-be-spawned task is allocated.

GPU CUDA thread team

Kokkos' CUDA task thread team is a CUDA warp of threads, thirty two threads that share an instruction cache and execute instructions "lock step." Given lock step execution the thread team is always synchronized and thus explicit barriers are not necessary. Instead our challenge is to provide thread team (warp level) collective reduce and scan operations that accomodate (1) a limited amount of GPU high-performance intra-team shared memory and (2) preparing for team and vector level data parallelism within tasks.

To reduce consumption of shared memory we make extensive use of CUDA *shuffle* operations to implement team collective reduce and scan operations. CUDA shuffle operations allows a thread to communicate data residing in its registers to other threads within the same warp. These operations can be a faster alternative to using shared memory when threads within a warp need to communicate data.

Kokkos' thread team data parallel execution policy supports two levels of nested parallel operations: team level and vector level. On CPU architectures team level parallel operations are split among hardware threads within the team and vector level parallel operations are attempted to be mapped onto special vector instruction hardware. On GPU architectures a CUDA thread block is divided into M team member "threads" that perform team level operations and V "vector" lanes that perform vector level operations. Thread team tasks are mapped on warps instead of full thread blocks so "thread" and "vector" division occurs within individual warps as opposed to full thread blocks, and thus $32 = M * V$. For example, in Figure reffig:cuda:members a 32 thread GPU warp is divided into four team member "threads" each with eight "vector" lanes.

Vector level, or *intra-member*, parallel operations are partitioned "vector lane" GPU threads and team level, or *inter-member*, parallel operations are partitioned "team member" GPU threads, as depicted in Figure 3.9. Data parallel operations at the vector or team level are structurally identical, with the difference isolated to which threads participate in the operation.

GPU CUDA driver

Kokkos' CUDA task DAG execution driver is dispatched to the GPU whenever a user calls the `wait` function on the task policy. The driver runs a CUDA worker function on each

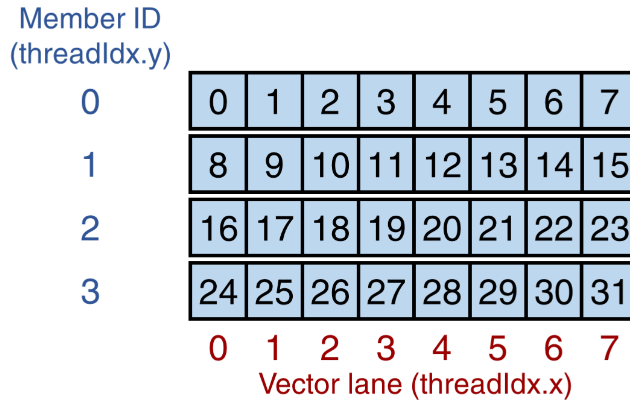
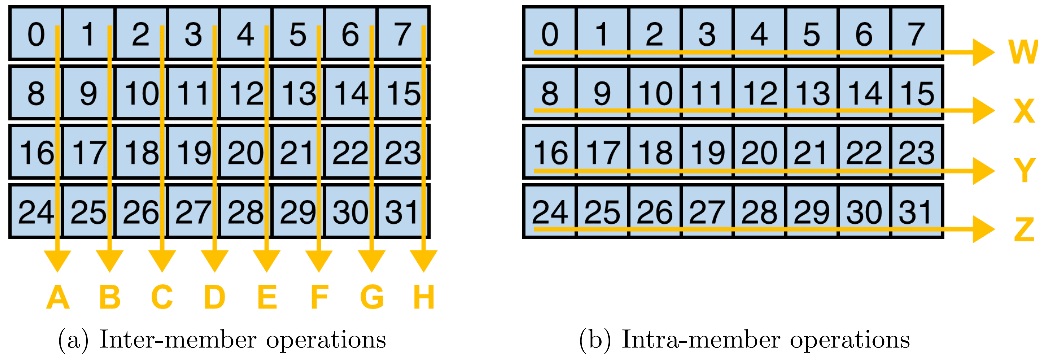


Figure 3.8: A 32 thread GPU warp is divided into M *thread team members* V *vector lanes* where M is denoted by CUDA's `blockDim.y` value, V is denoted by CUDA's `blockDim.x` value, a GPU thread's team member rank is denoted by CUDA's `threadIdx.y` value, and a GPU thread's vector lane is denoted by CUDA's `threadIdx.x` value. GPU threads grouped within vector lanes are adjacent for memory accesses; *i.e.*, they most performantly access a cache lines as a group.



(a) Inter-member operations

(b) Intra-member operations

Figure 3.9: Team and vector levels of parallel operations within a task are assigned to GPU threads according to the thread team member and vector lane division of the GPU warp.

GPU symmetric multiprocessor (SMP) where these worker functions loop until all executing and ready tasks are complete. Conventional GPU functions execute once on each thread within a CUDA *grid-block* hierarchy. As SMPs are available the GPU's built-in scheduler maps blocks within a grid to SMPs and runs the CUDA function on that block. In contrast a task DAG worker function is "parked" on each SMP until execution of the task DAG is complete. This GPU algorithm strategy is often referred to as using *persistent threads*.

As with the OpenMP implementation only the team lead (one hardware thread withing the warp) attempts to pop a task from a ready queue, and then uses shuffle operations to broadcast this task's pointer to the rest of the warp. If a thread team task then the entire team calls the task's execution function; otherwise only the team lead calls the execute

function and all other team member threads remain idle. When the task's execute function returns the team lead either reschedules the task if it respawned or schedules the completed task's wait queue.

A shared "executing plus ready" task count is used in the CUDA implementation to insure that all GPU threads exit the worker loop together.

3.4 Portable and Thread Scalable Memory Pool

Allocation and deallocation of task and when-all entity memory must be scalable with respect to the number of hardware threads concurrently executing the dynamic task DAG. Of particular concern is the GPU architecture with $O(1000)$ of concurent hardware threads. To meet this requirement we developed a portable, thread scalable memory pool allocator which was designed to be performant on GPU architectures with simple cores and large concurrency, and with acceptable performance on other architectures. Ideas from existing GPU compatible allocators ScatterAlloc [5, 6], Halloc [7], and Xmalloc [8] were leveraged in the development of our memory pool allocator.

In our dynamic task DAG use case, only one thread within a thread team allocates or deallocates task or when-all entities. Thus for the CUDA thread team, only one thread within the warp ever performs an allocation or deallocation. Our allocator assumes this behavior to allow a simpler and better performing design and implementation. We believe that requiring at most one thread per warp to perform allocation or deallocation is not overly constraining for more general use cases; and is worth the performance benefits.

The memory pool allocator interface given in Figure 3.10, has only four functions: the constructor, allocate, deallocate, and is_empty. An allocator can only be constructed outside a parallel operation; *i.e.*, by the main process. The allocate, deallocate, and is_empty functions can only be called within a parallel operation and may be interleaved as needed.

```
1  template <typename Device>
2  class MemoryPool {
3    MemoryPool( const Device::memory_space & memspace,
4                size_t total_size, size_t log2_superblock_size = 20 );
5
6    void * allocate( size_t alloc_size );
7
8    void deallocate( void * alloc_ptr, size_t alloc_size );
9
10   bool is_empty();
11 };
```

Figure 3.10: The memory pool allocator API has only four simple functions: constuctor, allocator, deallocator, and is_empty query.

The memory pool constructor is given a memory space from which the pool of memory is allocated, the total size of this pool, and an optional specification of the implementation's superblock size in bytes. In most cases, the default superblock size of 2^{20} bytes is acceptable.

The allocated pool of memory is reserved for exclusive use by the allocator until the allocator object is destroyed. The allocate function takes an allocation size and returns either a pointer to the memory allocated from within the pool or NULL indicating no memory is available. The deallocate function takes a pointer to the memory to deallocate and the allocated size of that memory. The is_empty function returns false if the allocator still has memory available to allocate and true otherwise.

3.4.1 Design and Implementation

Hierarchical partitioning of memory

At construction the memory pool allocator acquires a single contiguous chunk of memory from the input memory space and retains this chunk until the allocator is destroyed. This chunk of memory is hierarchically partitioned into superblocks, pages within superblocks, and blocks within pages. Each allocation returned by the allocate function occupies exactly one block in this hierachy.

Size of superblocks, pages, and blocks

The superblock size is specified at construction, must be a power of 2 between 2^{11} and 2^{31} bytes, and has a default value of 2^{20} bytes. A page normally contains 32 blocks, but it may contain fewer blocks if the block size is so large that the superblock can contain only a single page with fewer than 32 blocks. Individual blocks are 2^n bytes with a minimum size of 64 bytes and maximum size equal to a single superblock.

Active superblocks and the active superblock array

Superblocks that have allocated blocks are assigned to a single block size and contain blocks of a uniform size. Thus, a superblock will never have simultaneous allocations of multiple block sizes.

Allocations are only initiated on *active* superblocks. The memory pool allocator maintains an array of superblock IDs that indicates the active superblock for each block size. When an active superblock becomes full during an allocation, it is swapped for a superblock that has space available. Normally, allocations complete while the superblock is still active. However, it is possible for allocations to complete on a superblock after it has been swapped and is no longer active. To conserve memory the block sizes are initialized without active superblocks by setting the entries in the active superblock array to the special INVALID_SUPERBLOCK value.

Superblocks are only assigned to a particular block size on the first allocation requiring

that block size. Once a block size is required, there will always be a superblock assigned for that block size in the active superblock array. Removing an empty superblock from the active superblock array would require additional concurrency considerations, may require a locking strategy, and is likely to degrade performance. We chose to avoid this complication and potential performance degredation for our initial implementation of the memory pool allocator.

Meta-data

The memory pool allocator maintains meta-data for each block size. This meta-data includes the number of blocks per superblock with that block size, the number of pages per superblock, the *full threshold* for a superblock, and the *full threshold* for a page. Block size meta-data is initialized when the memory pool allocator is constructed and remains constant for the lifetime of the allocator.

Each superblock also has meta-data. Superblock meta-data includes the log_2 of the block size currently assigned to the superblock, the number of full pages, the number of empty pages, and whether the superblock is currently active. This information is initialized whenever a superblock is assigned to a blocksize and is updated as allocations and deallocations occur.

Limiting (spin) locks and other concurrency contentions

The memory pool allocator is designed to limit locking and other concurrency contentions during allocations and deallocations in order to achieve good performance. Each superblock has a bitset that maintains the allocated status of each of its blocks. Blocks are allocated and deallocated by using atomic boolean operations to set and unset the blocks' corresponding bits in the bitset. Since the bitset's word size equals the number of entries in a page, contention only happens in the atomic boolean operations when threads attempt to concurrently allocate or deallocate from the same page. Thus, the allocator attempts to scatter allocations for a particular block size across the pages of the active superblock to limit contention on the bitset. Contention in the bitset during deallocations cannot be controlled by the allocator and is dependent on the pattern of concurrent calls to deallocate. However, deallocation has had good performance in all testing so contention seems to not be an issue.

Locking is only required when swapping out a full superblock during an allocation, as described in Section 3.4.2.

Full threshold

To be able to perform the necessary state transitions of superblocks, the allocator needs to know when a superblock is full or empty. A superblock is full if a certain threshold of its pages

are full. A page is full if a certain threshold of its blocks are allocated. Thresholds of 80% for superblocks and 87.5% for pages yielded good performance with the CUDA implementation. We define a fullness threshold less than 100% so that concurrent attempts to allocate blocks that began below the threshold may succeed when the aggregate of those attempts exceed the threshold. This threshold strategy is intended to balance the potential for wasting superblock memory with reducing latency of waiting for superblocks to be swapped during allocations.

Each superblock has counters for the number of full and empty pages which must be incremented or decremented as pages become and stop being full and empty. The allocator determines the fullness of a page by counting the number of bits set in the value returned by the atomic boolean operations used to reserve / unreserve a block. This *popcount* is performed by a single hardware instruction and has good performance. The two-level fullness scheme is more performant than atomically updating a single counter for the number of blocks reserved per superblock due to decreased contention when updating the block reservation word. The full and empty page counters do cause some contention for concurrent allocations and deallocations in the same superblock. However, this contention is much less because the full and empty page counters are each only atomically updated during, on average, 1/32 of the allocations and deallocations.

Memory consumption overhead

The memory pool has several ways in which unused memory becomes unavailable for allocation including *fragmentation* and wasted memory. As memory is allocated and deallocated the memory pool may have multiple superblocks with the same block size that are only partially used; *i.e.*, superblock memory may become fragmented causing more superblocks to be used that are needed for the current allocations. Fragmentation also occurs when an allocation request is fulfilled with a block size that is larger than the requested size. These two sources of fragmentation occur in most memory allocators. In addition, superblock memory is wasted whenever an active superblock is swapped out before it is completely full. The remainder of the memory pool allocator's memory consumption overhead is quite small. The largest of this overhead comes from the bitset representing the allocation status of each block. This bitset is approximately .2% of the size of the initially allocated block of memory. The remaining memory overhead, which is of negligable size, is consumed by block size and superblock meta-data.

3.4.2 Superblock States

Superblock state transitions, such as swapping out a (nearly) full superblock for a non-full superblock, sometimes require spin locks for "atomic" updates to multiple variables. As such, we carefully design superblock states and state transitions to mitigate the risk of concurrency race conditions in the state transition operations. While not formally proven to be race-free, extensive testing provides empirical evidence that we have succeeded in mitigating this risk

for all but one extremely unlikely scenario.

Superblock states

A superblock can be in one of four mutually exclusive states of empty, active, partially full, and full; as illustrated in Figure 3.11. Each non-empty superblock is assigned to blocks of a particular size, and only one superblock is active for a given block size. An empty superblock has no blocks currently allocated and does not have an assigned block size. A partially full superblock has at least one block allocated but has not yet reached its *full threshold*. A full superblock has enough blocks allocated that it has exceeded its *full threshold*.

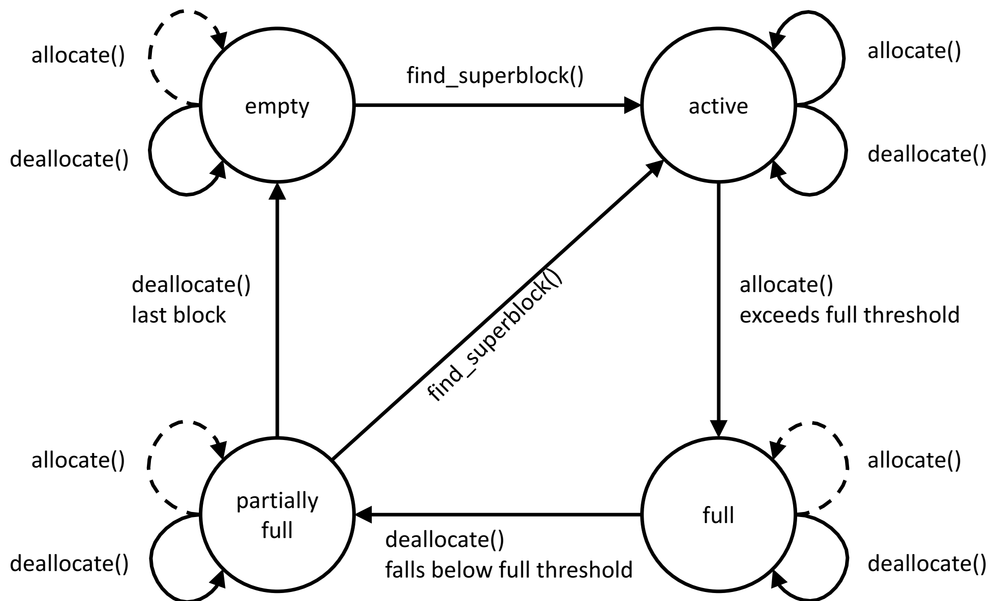


Figure 3.11: State diagram for superblocks. Solid line transitions are normal transitions. Dotted line transitions indicate the call to allocate() initiated on an active superblock but the superblock changed state due to other threads before the allocation finished.

Tracking superblocks

Superblocks in the active state are kept track of in two ways. First, their ID occurs in the active superblock array. Second, each superblock has a boolean flag in its meta-data indicating if it is active.

Superblocks in the empty and partially full states are tracked using bitsets, where a set bit denotes the corresponding superblock is in that state. The bitset for the partially full state is 2D with dimensions of number of superblocks and block sizes. A superblock can only be set once in the bitset; *i.e.* it can only be set for one block size. The 2D bitset trades extra space to speed up searches by find_allocate for partially full superblocks of a given

block size. Superblocks in the full state need not be explicitly tracked with a bitset since a superblock can only change from the full state to partially full state during a deallocation operation, which can easily find the full superblock containing the deallocated block using pointer arithmetic.

Normal state transitions

Figure 3.11 shows the state transition diagram for superblocks. Superblocks only transition between states during calls to the public allocate and deallocate functions or during a call to the internal find_superblock function. The find_superblock function is called by allocate when the active superblock is full and needs to be replaced. It finds a new superblock for the given block size and swaps it with the current active superblock for that block size.

The solid lines in Figure 3.11 denote normal state transitions. All superblocks start in the empty state when the memory pool allocator is constructed. The only state an empty superblock can transition to is the active state. This happens during either the first allocation for a particular block size or during an allocation that makes the current active superblock full.

In the active state, both allocations and deallocations are expected, and normally these do not cause a state change. The only state an active superblock can transition to is the full state. This happens when an allocation on an active superblock causes its number of internally allocated blocks to exceed its *full threshold*.

When a deallocation occurs on a full superblock, that superblock will transition to the partially full state if its number of allocated blocks falls below its *full threshold*. When a deallocation occurs on a partially full superblock, that superblock will transition to the empty state if its last allocated block is deallocated. A partially full superblock can also be chosen by the find_superblock function to replace the active superblock with the same block size, which will transition the partially full superblock to the active state. A deallocation is possible on an empty superblock if a user erroneously calls the deallocate function for memory that was previously deallocated and has not been reallocated.

Uncommon state transitions

The dotted lines in Figure 3.11 denote state transitions that may occur when an allocation was initiated on an active superblock but that superblock's state was changed by another thread before the allocation could complete. Allowing these allocations to complete even when the superblock's state changes helps to hide the latency of switching out the active superblock. Chances are good that an allocation will complete on a superblock that has transitioned from active to full due to our threshold strategy. The chance of an allocation occuring on a partially full superblock is much lower since that superblock must first transition from active to full and then from full to partially full before the allocation can

complete. These allocation superblock states are correctly managed in the current design and implementation.

The chance of an allocation occuring on an empty superblock is extremely remote. A superblock must transition from active to full, then from full to partially full, and then from partially full to empty before the allocation can complete. While we have not encountered this state during exhaustive testing, the state is theoretically possible and will be addressed in future work.

3.4.3 Allocation

The allocation function

- determines a block size for the requested allocation,
- obtains the active superblock for that block size,
- attempts to find, claim, and return a free block from the active superblock, and
- if this attempt fails the allocation function must obtain a new active superblock.

Given a requested allocation size, the smallest 2^n block size that is greater than or equal to the requested allocation size is chosen. An active superblock corresponding to the chosen block size is obtained for the allocation. The active superblock may be locked if it is undergoing a state transition; *i.e.*, is being swapped by another thread. If the active superblock is locked, the allocation function spin waits until the lock is released.

The allocation function chooses a starting page in the superblock and attempts to reserve any free block in the page. This involves finding a free block in the page and attempting to atomically set the block's corresponding bit in the bitset and repeating until a block is reserved or the page has no available blocks. When there are no available blocks in the page, the remaining pages of the superblock are linearly searched for a free block. If the reserved block is the first from an empty page, then the superblock's empty page count is decremented. If reserving a block puts the page above the full threshold, then the superblock's count of full pages is incremented. If incrementing the full page count exceeds the superblock's full threshold or an exhaustive search for a free block fails, then the find_superblock function is called to obtain a new active superblock.

Performance

Performance of the allocate function is largely dependent on the number of concurrent threads attempting to allocate memory from the same page. Thus, the method for choosing a starting page in the superblock is critical for performance. We first tried maintaining a

page index for each superblock and atomically incrementing it for each allocation, but incrementing this index became a performance bottleneck. We next tried using a random number generator, leading to even worse performance. We settled on using the value of the processor clock or cycle count register. Querying this value from a system register is extremely fast, and it provides enough variability in choosing a starting page to provide acceptable performance.

Failure to allocate

An allocation request larger than the superblock size to the allocate function fails and returns `NULL`. When an active superblock becomes full and there are no superblocks available to swap it out with, the superblock remains active. As long as there are no superblocks available to swap out with the active superblock, allocations can continue until the active superblock is completely full. At that point the allocation function fails and returns `NULL`.

GPU considerations

Our expected use cases have only one thread per GPU warp calling the allocation function. As such, we optimized the memory pool allocator design accordingly. For example, we need not protect against deadlocks that would occur if one thread within a warp locks a superblock while another thread within the same warp spin waits for that lock to be released. Redesigning to relax this constraint is possible by coalescing all allocation requests from the same warp to a single request (*e.g.*, this is supported by Xmalloc) but would increase memory and runtime overhead.

3.4.4 Switching Superblocks

During allocations the active superblock for a block size needs to be swapped out when it becomes full. The find_superblock function searches for an available superblock and then swaps it with the active one. To hide the latency required to swap out the active superblock, the full threshold for a superblock is chosen to be smaller than when a superblock is completely full. This allows allocations to continue in a superblock as it is being swapped out from being the active superblock.

The first step to swap out an active superblock is an attempt to acquire the lock on the block size's entry in the active superblock array. This lock is only for the specific block size. Allocations from all other block sizes will proceed unimpeded. A thread attempts to lock the entry in the active superblock array by performing an atomic compare-and-exchange (CAS) operation to replace the entry with the special value SUPERBLOCK_LOCK. Thus if multiple threads "race" to concurrently swap out the same superblock, only one thread will succeed. The unsuccessful threads reread the entry in the active superblock array until the

lock is released, and then accept the new non-lock value as the new active superblock.

The thread that successfully acquires the lock first searches for a superblock of the same block size that is in the partially full state. If a partially full superblock is not found, the thread searches for a superblock in the empty state. If a superblock is found from either state, the thread removes it from the partially full or empty bitset, sets the superblock's meta data appropriately, and makes that superblock active by performing a CAS to replace the lock value in the active superblock array with the new superblock's ID. If the thread cannot find a new superblock, it leaves the original superblock as the active superblock by performing a CAS to put its ID back into the active superblock array.

3.4.5 Deallocation

Deallocation is simpler than allocation. The first step to deallocate a block is determining the given memory location's superblock and page. Because the memory pool allocator holds a single contiguous chunk of memory, these can be easily calculated with pointer arithmetic. If the memory location is not contained within the allocator's memory, then deallocation fails and no action is taken. Otherwise, an atomic Boolean operation is performed on the bitset to free the block. If the block is already free the Boolean operation will fail without changing the bitset.

If the deallocation causes the containing page to become empty, then the superblock's empty page counter is incremented. If the deallocation results in the containing page dropping below the full threshold, then the superblock's full page counter is decremented.

3.4.6 Empty

The memory pool allocator is empty if it cannot perform another allocation. Determining if the allocator is empty is non-trivial and is only valid at a particular instant in time. If multiple threads are allocating and deallocating memory when a call to is_empty is made, then the answer returned may be out of date by the time it is checked. This is a universal problem for multi-threaded data structures.

Checking if the allocator is completely empty requires linearly scanning the bitsets of the superblocks, which is expensive. Instead, the allocator reports it is empty if all of its superblocks have reached their full threshold. Thus, the answer reported by the is_empty function is approximate. If is_empty reports true, the memory pool allocator might still be able to allocate. For example, an active superblock may be above the full threshold but still not completely full.

An allocator may not have superblocks to allocate blocks of size K but still have superblocks available for blocks of another size. In this case the allocator will report it is not empty; however, allocations of block size K will fail.

The true answer as to whether a memory pool allocator can or cannot allocate memory of a given size is determined by an attempt to allocate that memory.

Chapter 4

Integration with the Multithreaded Graph Library (MTGL)

The MultiThreaded Graph Library (MTGL) is designed for shared memory platforms and inherits multithreading primitive operations through an abstraction layer. This layer provides a consistent interface for library users. An algorithm can be be written once, then run in various multithreaded environments. The library was originally designed to export a generic API for the Cray XMT [9], then was adapted to leverage commodity multicore machines via qthreads. Through this LDRD, it has evolved to integrate Kokkos primitives in order to support both OpenMP and CUDA back ends.

We begin with the assumption that graphs will be instantiated entirely in device memory, then relax this unrealistic assumption to a more sustainable one: the vertices and associated property maps can live in device memory, while edge information will live in normal memory.

4.1 Introduction to the MTGL

The Boost Graph Library [10] (BGL) is a single-threaded set of abstractions that supports convenient graph computation. The library makes heavy use of advanced C++ features and provides the motivation for both the Parallel Boost Graph Library [11] (PBGL), a distributed-memory variant, and the MultiThreaded Graph Library [12] (MTGL), a single node, multithreaded variant. The latter is separated from the Boost libraries since these are extensive, single-threaded, and may not be conveniently portable to special platforms.

All of these libraries are written using *generic programming*. This programming style separates algorithm from data structure. In C++, it is accomplished via templated functions. We will show detailed examples of this style in Section 4.3. The central idea of the style is that code should be reusable despite changes in underlying data structure, so access to the latter is completely abstracted away from the application programmer.

These libraries also provide convenient access to graph *property maps*, which associate attributes with graph objects such as vertices and edges. In the BGL, these maps can be *internal* or *external*. Internal maps are tightly associated with graphs at graph definition time, and typically store attributes that cross algorithm boundaries such as vertex id, edge

weight, and/or vertex "distance" (which might be in terms of number of hops from some search root or a true distance that takes edge weight).

External property maps typically store attributes that weren't anticipated or justified at graph creation time. Some examples might include algorithm-specific quantities such as vertex rank in a PageRank algorithm or network flow/capacity in a flow algorithm. The distinction between internal and external properties takes on special importance when considering dynamic graphs and/or graphs that may have to be transferred to a special device prior to being manipulated.

The MTGL was originally written to run on the Cray XMT supercomputer, but was later generalized to run on multicore workstations [13] via qthreads [1] or OpenMP. In this document, we discuss the Kokkos-inspired API for the next major release of the MTGL: 2.0, expected in 2017.

4.2 Graph abstractions in the MTGL

The MTGL is a generic software library, and therefore does not depend on any one data structure. Rather, the API to access graph elements such as vertices, edges, and their related properties is data structure-agnostic. When faced with a new underlying data structure, the MTGL developer implements this API. Existing library algorithm code will then be applicable to the new structure. We describe this API in Section 4.3.

Furthermore, the abstractions of MTGL 2.0 described in this document require *C++11*, which enables advanced language features such as temporary ("*lambda*") functions in order to leverage the Kokkos software effectively.

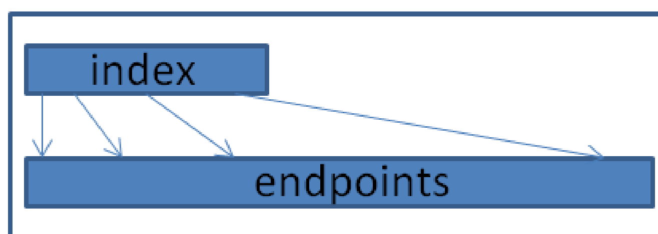


Figure 4.1: The compressed sparse row graph data structure

To facilitate discussion, we consider a familiar graph data structure: the compressed sparse row (CSR) graph. At its most basic level, this structure consists of two arrays: an array of indices and an array of endpoints. Implementations such as the default MTGL *compressed_sparse_row_graph* also store other information such as the source points - to speed up iterating over edges - and vertex / edge properties. We note that more memory-frugal versions of this data structure can be used with the MTGL algorithm functions.

The CSR structure is immutable: vertices and edges are not added or deleted once the graph has been initialized. An example of this structure is instantiated in the MTGL by the *compressed_sparse_row_graph* C++ class. This class is the easiest complete example implementation of the graph API.

A primary alternative to the CSR representation is an adjacency list representation, in which each vertex has its own adjacency structure. The MTGL instantiates this idea via the *adjacency_list* class. This is a dynamic structure in which vertices and edges can come and go. There are more complicated variants of adjacency lists such as the STINGER structure of [14]. In the latter, adjacency lists are partitioned into linked lists of blocks.

The MTGL API provides consistent ways to access all of these structures conveniently while preserving good multithreaded performance at runtime. Before going into this API, however, we describe the concept of graph *adapters*. This concept is inherited from its use in the Boost Graph Library. The idea is that transformations of a graph (such as the *transpose*, in which edge directions are reversed) or extraction of subgraphs must produce results that are themselves first-class graph objects implementing the full MTGL API.

Adapters in the MTGL are typically passed to algorithms by reference. For example, the consider the signature of the library function *find_triangles* in Figure 4.2.

```
1 // find_triangles signature
2 template <typename Graph, typename Visitor>
3 void find_triangles(Graph& g, Visitor& tri_visitor);
```

Figure 4.2: a typical MTGL algorithm signature

The first argument is a reference to an object implementing the MTGL graph API and the second argument is a reference to a *visitor* object that includes a user-defined method to process each triangle as it is enumerated.

While the current MTGL pattern is to pass these objects by reference, we have had to revisit this decision when considering Kokkos/MTGL integration. Kokkos views are always passed by value, and a deep copy happens only upon explicit programmer request. In Section 4.3 we will discuss the resulting changes to the MTGL library function calling pattern.

The library function `find_triangles()` and others accept adapters as well as graphs. For example, the code segment in Figure 4.3 shows how we would create a subgraph of graph g and enumerate its triangles. The library routine `find_triangles` is completely generic,

```
1 // calling an MTGL algorithm (a Graph g exists already)
2 find_triangles(g, tri_visitor);
3
4 // calling an MTGL algorithm on a subgraph (identical function calling convention)
5 subgraph_adapter<Graph> subg(g);
6 init_vertices(bool_vertex_mask, subg);
7 find_triangles(subg, tri_visitor);
```

Figure 4.3: calling an MTGL algorithm

thanks to the the graph API we will discuss in Section 4.3.

4.3 The MTGL API

We begin our API discussion with the types and objects used by the MTGL library algorithms. Any data structure must provide these types, descriptors, and iterators - perhaps via a user-defined adapter that controls access to the data structure internals. Vertex and edge descriptors (handles) must be comparable (vertex to vertex, edge to edge).

4.3.1 Sizes, descriptors, and iterators

A *vector iterator* is analogous to an STL random access iterator. However, the MTGL API functions typically use a single instance of such an iterator rather than the STL-style {begin, end} pair. This is an artifact of the distinctly non-STL programming patterns of the Cray XMT. Table 4.1 shows the C++ type names associated with MTGL sizes, descriptors, and iterators.

Sizes	
size_type	The type that represents sizes used to index vertex and edge arrays
Descriptors	
vertex_descriptor	The handle used to refer to a vertex
edge_descriptor	The handle used to refer to an edge
Vector Iterators	These provide random access to vertices and edges
vertex_iterator	Random access iterator through all vertices
edge_iterator	Random access iterator through all edges
adjacency_iterator	Random access iterator through all adjs of one vertex
in_adjacency_iterator	Random access iterator through all in-adjs of one vertex (if avail.)
out_edge_iterator	Random access iterator through all incident edges leaving a vertex
in_edge_iterator	Random access iterator through all incident edges pointing to a vertex
Thread Iterators	These provide incremental access to vertices and edges
thread_vertex_iterator	Incremental access iterator through all vertices
thread_edge_iterator	Incremental access iterator through all edges
thread_adjacency_iterator	Incremental access iterator through all adjs of one vertex
thread_in_adjacency_iterator	Incremental access iterator through all in-adjs of one vertex (if avail.)
thread_out_edge_iterator	Incremental access iterator through all incident edges leaving a vertex
thread_in_edge_iterator	Incremental access iterator through all incident edges pointing to a vertex
Categories	These describe global attributes of the graph
directed_category	{undirected, directed, or bidirectional}.
iterator_category	the graph either supports vector iterators, thread iterators, or both

Table 4.1: MTGL Types, Descriptors, Iterators, and Categories

Thread iterators are abstractions that provide iteration through potentially more complicated data structures. These are typically adjacency list-based, such as STINGER [14]. Unlike vector iterators, there is no random access. The iterator object is constructed with an iteration starting point, and increment operators advance the iterator. This still allows efficient parallelism and reduces to simple array access in the case of CSR graphs.

4.3.2 Atomics and concurrency

The MTGL API also provides abstractions to access certain atomic instructions and locking primitives. The `mt_incr` and `mt_cas` calls reduce to hardware atomics for int-fetch-add and compare-and-swap, respectively - if available. If not, they provide software wrappers for these features. Massively multithreaded applications expect fine-grained locking at the word level, and the MTGL `mt_readfe` and `mt_write` abstractions provide access to these hardware primitives - or else simulate them in software.

4.3.3 Generic functions

The comprehensive list of the MTGL generic functions is evolving, but we will provide several examples below. Once a graph object has been declared (no matter its type), generic functions such as the following can be applied. These are all templated functions, but the template syntax is not shown.

- size_type num_vertices(const Graph& g); // return the number of vertices
- size_type num_edges(const Graph& g); // return the number of edges
- vertex_iterator vertices(const Graph& g); // return an iterator over the entire vertex set
- vertex_descriptor source(const edge_descriptor& e, const Graph& g); // return the first vertex in edge e.
- vertex_descriptor target(const edge_descriptor& e, const Graph& g); // return the second vertex in edge e.
- etc.

We refer the reader to `https://software.sandia.gov/trac/mtgl/wiki/GraphAPI` for the current complete list.

4.3.4 Basic functionality

In this section we show how to write programs in the MTGL that create graphs and traverse their vertex and edge sets. The full code for these examples can be found in two files, each with roughly 400 lines of code. They are located on a branch at the time of this writing, but eventually will be in the *mtgl/test* directory of MTGL 2.0:
`https://...mtgl/browser/branches/noxmt/test/test_algorithm_graph.cpp`.
`https://...mtgl/browser/branches/noxmt/test/test_parallel_graph.cpp`.

This section is intended to serve as a quick start once the MTGL is compiled This code will work with OpenMP or will leverage a GPU device via CUDA, depending on the way it is compiled. The device-specific details are abstracted into the MTGL library include files, which in turn rely on the Kokkos API to leverage the desired execution environment.

Library algorithms such as connected components, PageRank, etc., will be considered in Section 4.4.

Basic MTGL include files

```
1 // MTGL basic includes and namespace
2 #include <mtgl/mtgl_test.hpp>
3 #include <mtgl/compressed_sparse_row_graph.hpp>
4
5 using namespace mtgl;
```

Figure 4.4: basic MTGL includes and namespace

Most MTGL programs will begin by including two include files: `mtgl_test.hpp`, which provides basic graph creation and I/O functions, and a file containing the implementation of the basic graph data structure API depicted in Table 4.1. In our case, we include *compressed_sparse_row_graph*.

Starting out: typedefs and graph initialization

Figure 4.5 shows the header information of a typical MTGL program, which remains unchanged from the original MTGL API - yet now gives us access to Kokkos-enabled functionality. In the case of a CUDA back end, *mtgl_initialize* ignores the *num_threads* argument and the *create_test_graph* generic function abstracts away device-specific graph creation details. The only difference in such details between OpenMP and CUDA compilation at the time of this writing is that OpenMP supports both graph generation and reading (multiple formats), while the CUDA back end supports only the reading of binary graphs. The complete version will offer equivalent support via both back ends.

When building the MTGL with the Kokkos/CUDA "device," our default mode is "unpinned," which means that entire graph and its property maps will reside on the GPU.

```
// A typical test program (header information)
int main(int argc, char* argv[])
{
  typedef compressed_sparse_row_graph<directedS> Graph;
  typedef typename graph_traits<Graph>::size_type size_type;
  typedef typename graph_traits<Graph>::vertex_descriptor vertex_descriptor;
  typedef typename graph_traits<Graph>::edge_descriptor edge_descriptor;
  typedef typename graph_traits<Graph>::vertex_iterator vertex_iterator;
  typedef typename graph_traits<Graph>::edge_iterator edge_iterator;

  unsigned num_threads = init_test(argc, argv);
  mtgl_initialize(num_threads);

  Graph g;
  create_test_graph(g, argc, argv);
```

Figure 4.5: a typical MTGL test program: header information

In Kokkos lingo, this means that the *execution_space* of the Kokkos Views that underly the graph and its property maps will be CUDA. We also offer a "pinned" mode, in which the vertices and vertex property maps reside on device, but the edges and edge property maps must be loaded onto device from the host on demand. Most of this document treats the default unpinned mode, but we will discuss our preliminary results in pinned mode in Section 4.7.

Declaring iterators and property maps

```
// Declaring MTGL iterators and property maps
size_type order = num_vertices(g);
size_type size = num_edges(g);

vertex_iterator viter = vertices(g);
edge_iterator eiter = edges(g);

vertex_property_map<Graph, long> vprop(g);
vertex_property_map<Graph, long> vprop2(g);
edge_property_map<Graph, long> eprop(g);
edge_property_map<Graph, long> eprop2(g);
```

Figure 4.6: declaring MTGL graph iterators and property maps

Figure 4.6 shows how to call generic functions to retrieve the numbers of vertices and edges, and to obtain iterators through the vertex and edge sets. It also shows the declarations of vertex and edge property maps. Such declarations appeal to the Kokkos API to define approprite views for the current memory space and execution environment. These details are hidden from MTGL developers and users.

Property maps are essential for graph algorithms, as these are needed store attributes such as distance in shortest path computations, rank in PageRank-like algorithms, etc. In this example, we have declared two vertex property maps and two edge property maps to help demonstrate common idioms in MTGL programming.

Initialization and use of property maps

In the original MTGL API (1.x), programmers could access individual properties within property maps. In MTGL 2.0, if we are using the Kokkos/CUDA device in unpinned mode then the graph and its property maps reside on the GPU device and must be accessed via special MTGL generic functions that trigger Kokkos functionality. Figure 4.7 shows several examples.

```
1  // In CUDA, access to the graph and its property maps must
2  // be done through MTGL functions
3
4    vertex_descriptor v = viter[0];
5    // these were ok in MTGL 1.x, but are errors with Kokkos/CUDA
6    vprop[v] = 39;
7    put(vprop,v,39);
8    long val1 = get(vprop,v,39);
9
10   // these work by invoking Kokkos' deep_copy to the correct execution space
11   deep_put(vprop,v,39);
12   long val = deep_get(vprop, v);
13
14 // These generic functions run in the correct execution space
15   fill_v(vprop, g, 1);
16   fill_e(eprop, g, 1);
```

Figure 4.7: MTGL property map access

Alternatively, we can use Kokkos parallelization primitives via MTGL counterparts. The latter also ensure that the computation happens in the correct execution space. Figure 4.8 shows an example. When using the Kokkos/CUDA device, the MTGL_LAMBDA macro defines KOKKOS_LAMBDA, which abstracts away device-level details of temporary function definitions.

```
1  // The MTGL forall_vertices and forall_edges generic functions take a
2  // functor argument and wrap the Kokkos parallel_for
3    forall_vertices(g, MTGL_LAMBDA (const vertex_descriptor& v)
4    {
5      vprop[v] = 1; // this runs in the device execution space
6    });
7
8  // Now suppose that we need to manipulate these properties on the host
9    typedef vertex_property_map<Graph, long> vertex_map;
10   typename vertex_map::host_property_map host_vprop = create_mirror_map(vprop);
11
12   deep_copy(host_vprop, vprop);  // now we have the property data on the host
13   for (size_type i = 0; i < order; ++i) host_vprop[viter[i]] = host_func(i);
14   deep_copy(vprop, host_vprop);  // and now they are back on device
```

Figure 4.8: MTGL property map access with Kokkos parallel primitives and via the host

Device computations on property maps

We provide device computation on property maps in the Kokkos style. First we define a functor, then we apply that functor to all of the vertex or edge properties. Figure 4.9 provides

an example:

```
// A property map functor needs a constructor and an operator(key)

  template <typename PropertyMap>
  struct for_func {
    typedef typename property_traits<PropertyMap>::key_type key_type;

    PropertyMap map;

    for_func(PropertyMap& m) : map(m) {}

    MTGL_INLINE_FUNCTION  // tells Kokkos to apply appropriate device directive
    void operator()(const key_type& k) const { map[k] = 2; }
  };

  ...

  // within another function, apply the functor to the property of each vertex
  for_func<vertex_map> ffv(vprop);
  forall_vertices(g, ffv);  // wraps Kokkos::parallel_for
```

Figure 4.9: applying a functor to an MTGL property map

We also provide classical parallel algorithmic primitives at a higher level of abstraction. Perhaps the most widely used of these is *parallel prefix*. Given a vector of n objects a and an associative operator $\circ$, parallel prefix computes all prefix sums $a_1 \circ \ldots \circ a_i$, for $1 \leq i \leq n$. Motivated by the mathematical term for this type of operation (also used to name a key operation of the Cray XMT compiler), we provide *linear_recurrence* functions to compute parallel prefix. Figure 4.10 shows an example:

```
// Precondition: vertex property map vprop is initialized to 1 for all v

  // edge property maps are manipulated the same way as vertex property maps
  forall_edges(g, MTGL_LAMBDA (const edge_descriptor& e)
  {
    eprop[e] = 1;
  });

  linear_recurrence_v(vprop, g);
  // now vprop[i] == i*(i+1)/2
  linear_recurrence_e(eprop, g);
  // now eprop[i] == i*(i+1)/2
```

Figure 4.10: computing parallel prefix on vertex and edge property maps using the *linear_recurrence* functions

The components used to build the *linear_recurrence* functions are flexible. For example, it is straightforward to replace the default *sum* operation with other binary operations. Examples of that detail, while be beyond the scope of this document, can be found in the MTGL (the simplest of which are the definitions of *linear_recurrence*).

The next common primitive is the classical *reduction*. Given a vector of n objects a and an associative operator $\circ$, the reduction operation computes a single result $a_1 \circ \ldots \circ a_n$. Usually this result is a scalar, but that need not be in general. Kokkos provides a flexible interface to reduction functionality, and we provide another layer of abstraction for MTGL

computation - especially on vertex and edge property maps. Figure 4.11 shows examples in both cases using lambda functions.

```
// precondition: vprop and eprop are all initialized to 1

  // Given graph g, return the sum of vprop[v] for all vertices v
  reduce_vertices(g, MTGL_LAMBDA (const vertex_descriptor& v, long& my_sum)
  {
    my_sum += vprop[v];  // my_sum is local; Kokkos joins these into ``sum''
  }, sum);

  // reducing over edges is similar
  reduce_edges(g, MTGL_LAMBDA (const edge_descriptor& e, long& my_sum)
  {
    my_sum += eprop[e];
  }, sum);
```

Figure 4.11: reductions over vertex and edge property maps using temporary functions

If the reduction operator is large, involved, reused, or otherwise worthy of a name other than "lambda," we also provide an API for defining standalone custom reduction operators and using them with the *reduce_vertices* and *reduce_edges* generic functions. We demonstrate this functionality in Figure 4.12.

```
// precondition: vprop and eprop are all initialized to 1

// This custom reduce operator shows how to reduce a key (property map
// entry) into a partially reduced result.
template <typename PropertyMap>
struct reduce_func {
  typedef typename property_traits<PropertyMap>::key_type key_type;

  PropertyMap map;

  reduce_func(PropertyMap& m) : map(m) {}

  MTGL_INLINE_FUNCTION
  void operator()(const key_type& k, long& my_sum) const { my_sum += map[k]; }
};
  ...
  ...
  reduce_func<vertex_map> rfv(vprop);
  reduce_vertices(g, rfv, sum);

  reduce_func<edge_map> rfv(eprop);
  reduce_edges(g, rfv, sum);
```

Figure 4.12: reductions over vertex and edge property maps using functors

4.4 MTGL library algorithms

The basic MTGL API is used to create a library of basic graph algorithms that will be called by application programmers. This library consists of an expanding set of optimized algorithms that can be composed gracefully. Application programmers are strongly discouraged from writing their own versions of existing library algorithms. If the current ones cannot be

composed to meet a set of needs, please contact the MTGL team (Greg Mackey, Jon Berry) with a feature/capabilities request.

4.4.1 Breadth-First Search

Perhaps the most basic of all graph algorithms is *breadth-first search* (BFS), which takes a set of seed vertices, then visits the vertices one hop away, then two hops, etc. The kernel operation of BFS, sometimes called *level set expansion*, or *expand one edge*, starts with all vertices at Level i (i-hops away from the seed set) and visits all vertices at Level $i+1$. The basic logic of this kernel is a nested loop, with the outer loop going over vertices in Level i, and the inner loop going over their adjacencies. The heavy-tailed degree distributions of many graphs makes this kernel a challenge to compute efficiently. One efficient approach is to merge the two loops via pre-computation, then have a single loop execute the entire computation. This is called "Manhattan loop collapse" in the compiler community. The MTGL library function *breadth_first_search* implements this operation efficiently. Note that we do not implement the direction-optimizing idea of Beamer, et al. [15] so that our generic code will work on directed graphs. A specialization for undirected graphs could be included at a future date.

The MTGL library algorithms all implement the *visitor pattern*, which allows application programmers to provide functors with user-defined reactions to various algorithmic events. User-defined visitor classes can contain arbitrary data, and therefore allow programmers to manipulate their own data structures (typically vertex and/or edge property maps) during the run of an algorithm. The visitor methods of functors passed to *breadth_first_search* are listed in Table 4.2. Note that these need not be defined every time; there is a default visitor object.

method	description
discover_vertex	react to the initial discovery of vertex u
examine_vertex	called just before traversing edges of u
examine_edge	called just before traversing incident edge e
tree_edge	called just after determining that the endpoint of e is on a new level
non_tree_edge	called just after determining that the endpoint of e is non on a new level
finish_vertex	called just after all incident edges on u have been traversed
filter	decide whether or not to traverse edge e
color_test	while edge $e=(u,v)$ is being traversed, decide whether v has been visited

Table 4.2: MTGL Breadth-first search visitor API

In order to run BFS, we declare a visitor object and pass it to a generic function that invokes the algorithm. Figure 4.13 gives an example in which the visitor is supposed to populate a property map with the BFS parent for each node.

The *bfs_parents* visitor object itself is defined in Figure 4.14. We inherit from the MTGL's

```
// calling the MTGL breadth_first_search algorithm

  // declare a vertex property map to hold bfs parent information
  typedef vertex_property_map<Graph, vertex_descriptor> VertexPredMap;
  VertexPredMap bfs_parents(g);

  // declare a visitor object that encapsulates this property map
  parents_bfs_visitor<Graph, VertexPredMap> pbv(bfs_parents);

  // now call the bfs
  vertex_descriptor source;
  ...
  breadth_first_search(g, source, pbv);

  // the result is that bfs_parents are now set; customized outcomes
  // are achieved by changing the visitor
```

Figure 4.13: calling the MTGL breadth_first_search

default_bfs_visitor, which provides stub implementations of all of the visitor methods (most of which optimized away during compilation). We override the *tree_edge* method to make an assignment to the property map.

```
// defining a bfs visitor.  this one sets ''parent'' attributes

// note that we inherit from default_bfs_visitor, which defines all of
// the visitor methods
template <typename Graph, typename ParentsMap>
class parents_bfs_visitor : public default_bfs_visitor<Graph> {
public:
  typedef typename graph_traits<Graph>::vertex_descriptor vertex_descriptor;
  typedef typename graph_traits<Graph>::edge_descriptor edge_descriptor;

  parents_bfs_visitor(ParentsMap& p) : bfs_parents(p) {}

  // and we just override one of them
  MTGL_INLINE_FUNCTION
  void tree_edge(const edge_descriptor& e, const Graph &g) const
  {
    vertex_descriptor u = source(e, g);
    vertex_descriptor v = target(e, g);
    put(bfs_parents, v, u);  // u is the bfs parent of v
  }

private:
  ParentsMap bfs_parents;
};
```

Figure 4.14: defining the visitor for MTGL breadth_first_search

In Section 4.5, we give performance results for this version of BFS in OpenMP and CUDA.

4.4.2 Connected Components

While breadth-first search stresses adjacency traversal from seed vertices, the classical Shiloach-Vishkin algorithm for connected components [16] stresses edge iteration. This algorithm

repeatedly processes all edges, then processes all vertices. Each loop updates property map values. There are no clear discrete events to motivate a visitor interface, so *shiloach_vishkin* is an example of an MTGL library algorithm that does not follow the visitor pattern. We also note that Shiloach-Vishkin is subject to hot spots in memory access. In large datasets, alternative algorithms may be preferable. However, this algorithm is quite serviceable in many contexts. Figure 4.15 shows the MTGL calling conventions.

```
1 // calling the MTGL library implementation of Shiloach-Vishkin
2
3   // make a property map to hold the result (the component of each vertex)
4   vertex_property_map<Graph, vertex_descriptor> components(g);
5
6   // and call the function
7   shiloach_vishkin(g, components);
8
9   // components are now set
```

Figure 4.15: finding connected components with the MTGL "shiloach_vishkin" algorithm

Two other algorithms currently running on both the Kokkos/OpenMP device and the Kokkos/Cuda device are PageRank (with in interface similar to that of *shiloach_vishkin*) and triangle (3-cycle) enumeration (with a visitor interface that exposes each triangle to the user for processing).

4.5 MTGL/Kokkos algorithm performance

We begin by evaluating the performance of simple MTGL primitives that wrap Kokkos primitives without ever defining a graph. These include `parallel_for`, `parallel_reduce`, and `parallel_scan` (parallel prefix).

4.5.1 Basic array processing

Table 4.3 shows the results for CPU and GPU runs of the MTGL test program *test_parallel* as it processes arrays of one billion 64-bit integer values. We note that there is a startup cost associated with the first call to `Kokkos::parallel_reduce` on an array of a given size. This first call is not timed, so the timings below do not account for this cost.

operation	OpenMP device (20 CPU)	Cuda device (GPU)	GPU speedup
parallel_for	0.1996s	0.0395s	5.05X
parallel_reduce	0.0977s	0.0528s	1.85X
parallel_scan	0.2965s	0.3586s	0.83X

Table 4.3: Basic Kokkos data parallelism through the MTGL interface

We see the speedups expected from Kokkos, including the lack of speedup for `parallel_scan`. GPU implementations of the classical two-pass algorithm from Blelloch [17] do show speedup over a single CPU [18], but not over 20 threads from our multicore machine. Recent work on parallel scan from Maleki, et al. promises to leverage GPUs more effectively [19]. Once this technique is integrated into Kokkos, we can expect to see these speedups through the MTGL interface.

4.5.2 Graph Iteration

Next we consider one further level of abstraction: iterating through sets of vertices and edges in MTGL graphs. We use the default *compressed_sparse_row_graph* as our test case. We run on a graph with approximately three million vertices and 117 million directed edges. Table 4.4 shows the results for CPU and GPU runs of the MTGL test program *test_parallel_graph* applied to this graph.

operation	OpenMP (20 CPU)	Cuda (GPU)	GPU speedup
forall_vertices	0.00062s	0.00014s	4.3X
forall_edges	0.02346s	0.00467s	5.02X
reduce_vertices	0.00043s	0.00029s	1.47X
reduce_edges	0.0139s	0.0062s	2.2X
scan_vertices	0.00072s	0.0024s	0.29X
scan_edges	0.0347s	0.0431s	0.80X

Table 4.4: Data parallelism and graph iteration

Note that we expect no speedup from the scan functions due to the algorithmic considerations explained above, especially in the case of scanning only three million vertices. Note also that the number of edges is roughly 10X smaller than the array size in Section 4.5.1, so we expect the edge processing times to be about one tenth of the array processing times for these three primitives. Tables 4.3 and 4.4 confirm this expectation, providing evidence that the overheads associated with MTGL/Kokkos graph iteration are negligible.

4.5.3 Graph algorithm performance

We compare the performance of several MTGL library algorithms on generated graphs and graphs from the open source. The former are R-MAT graphs [20], and the latter are social networks from open source. These include a snapshot of LiveJournal-2008 [21] and a snapshot of Orkut-2007 [22]. We postprocess the social networks to remove non-reciprocated relationships. The sizes of these datasets are shown in Table 4.5.

We also compare our results with Gunrock, a recently published graph library specific to GPU computations [23]. Gunrock claims to be competitive with all current GPU-exploiting

dataset	Vertices	Edges
LiveJournal-2008	5,363,201	28,477,243
com-Orkut	3,072,626	117,185,083
rmat-21	1,164,411	16,777,216
rmat-24	8,476,236	134,217,728

Table 4.5: Datasets for MTGL library algorithm experiments

graph libraries, and also with the best hard-coded GPU graph algorithms.

Table 4.6 shows our results for breadth-first search. This algorithm was carefully optimized using the techniques described in Section 4.6. Each entry in the table represents the average of three runs. Compiling the MTGL program with the Kokkos/Cuda device yields an average speedup of 3-4X when compared to the same code compiled with the Kokkos/OpenMP device. Our results are also comparable with those produced by Gunrock. We note that Gunrock seems to run slowly on small inputs in our trials, but we have not explored this observation.

input graph	OpenMP (20 CPU)	Cuda (GPU)	GPU speedup	Gunrock
LiveJournal-2008	0.00511s	0.00403s	1.27X	0.26802s
Orkut	0.77405s	0.20955s	3.69X	0.27726s
rmat-21	0.17199s	0.03895s	4.41X	0.1986s
rmat-24	1.2708s	0.2834s	4.48X	0.2882s

Table 4.6: MTGL breadth-first search performance

Table 4.6 shows our results for the MTGL implementation of PageRank [24]. This routine has also been optimized as per Section 4.6, but not as carefully as breadth-first search. However, the CUDA performance still dominates OpenMP. We were not able to make Gunrock run PageRank stably, so we do not include its results. Gunrock would perform ten times too many iterations, given a convergence tolerance.

input graph	OpenMP (20 CPU)	Cuda (GPU)	GPU speedup
LiveJournal-2008	0.0675s	0.0620s	1.09X
Orkut-2007	3.6832s	2.8657s	1.28X
rmat-21	0.2508s	0.1496s	1.67X
rmat-24	1.8680s	1.2330s	1.51X

Table 4.7: MTGL PageRank performance

For an example of a library algorithm not yet optimized for GPU computation, consider our implementation of Shiloach-Vishkin's connected components algorithm. Our results are included in Table 4.8. Note that without the customizations described in Section 4.6, the

algorithm runs on the GPU, but not with superior performance. We were able to run Gunrock on the smaller instances, and its GPU-optimized connected components routines beats OpenMP and our non-optimized Cuda version.

input graph	OpenMP (20 CPU)	Cuda (GPU)	GPU speedup	Gunrock
LiveJournal-2008	0.0709s	0.2060s	0.34X	0.5448
Orkut-2007	0.1460s	0.1481s	0.98X	
rmat-21	0.02507s	0.02310	1.08X	0.0508
rmat-24	0.3587s	0.3398	1.05X	0.6143

Table 4.8: MTGL connected components performance

4.6 Customizing MTGL algorithms to optimize Kokkos usage

The interface described in Section 4.3 is functional, and application programmers should be able to use it to write code that glues library components together into graph computations. However, MTGL library programmers will need another level of granularity in order to optimize library algorithms for GPU devices. The concepts of *warp* and *team* have are not exposed in the basic MTGL/Kokkos API, yet these concepts are necessary for optimal performance.

In particular, the fundamental strategy of blocking parallel iterations changes on a GPU. In CPU multithreaded programming, we typically have a choice between assigning threads iterations that process small, contiguous blocks of data, or assigning threads strides that touch much bigger sections of data. If cache size is generous and the number of threads is small, then it is often easier and more efficient so assign each thread work touching a small block of memory that other threads avoid.

However, on a GPU the number of threads is large and the shared memory per block of threads (warps and teams) is small. Furthermore, each thread in a warp runs the same set of instructions in lock step. These features typically mandate the strided approach to parallelism, in which a block of threads all process the same small block of memory simultaneously. Thread i is assigned memory location i within the block.

Our solution is to expose and document these GPU-specific primitives to the library programmer only. This section serves as an abstract for future documentation that will go into more detail.

MTGL library programmers can appeal to the Kokkos documentation to define *team_policy* objects. These will chunk up the work in accordance with the current device hardware. In the Kepler GPU we tested, there are 32 threads in a team. Figure 4.16, which abstracts the

file *mtgl/kokkos/kokkos_bfs.hpp* shows an example. shows the basics for manipulating teams.

```
1  // MTGL/Kokkos team policy
2
3    // total work: num_blocks * block_size, where
4    // block_size is the size of a chunk of iterations to be processed by
5    // one thread team.  For our example, block_size == 1024
6
7    // suppose that the device has 1024 threads in a "league" of 32 teams,
8    // each with 32 members. Threads 0..31 are in Team 0, threads 32..63 are
9    // in Team 1, etc.
10
11   // Kokkos::AUTO() will detect the number of threads in a warp
12   team_policy work(num_blocks, Kokkos::AUTO());
13
14   // pass the team_policy object to Kokkos::parallel_for
15   Kokkos::parallel_for(work, MTGL_LAMBDA (const member_type& member)
16   {
17     // Each thread in a team executes this code in sync
18
19     // league_rank will find which team (within the league of teams) I am in.
20     // This will identify the beginning of a block of iterations
21     size_type search_team_begin = member.league_rank() * block_size;
22
23     // and the team will bite off chunks of 32 iterations repeatedly until
24     // it has finished block_size iterations
25     size_type search_team_end = search_team_begin + block_size;
26
27     // as a thread, I find my stride (position within the team)
28     size_type thread_index = search_team_begin + member.team_rank();
29
30     // now find which MTGL object (vertex or edge I should process)
31     size_type vert_index = some_mtgl_function(...,thread_index)
32
33
34     while (search_team_begin < search_team_end) {
35         // do some work (each team member thread in sync)
36
37         // advance another 32 iterations and let the team process the next chunk
38         search_team_begin += member.team_size();
39         thread_index += member.team_size();
40     }
41   }
```

Figure 4.16: Team-level MTGL library coding

4.7 Processing graphs that don't fit on device

Kokkos offers a "pinned memory" option in which blocks of memory are allocated on the host, then brought into device via demand paging. This enables computations on datasets too large to fit on device, and we anticipate needing to rely on this feature or a double-buffering analogue for real applications.

We revised our breadth-first search code in order to leverage pinned memory, and in inital experiments, we saw about a 2X slow-down over pure device computation. However,

we were encouraged by near-linear weak scaling as we increased the graph size far beyond the size of device memory. We do not include these results here since they are deprecated. In the future we will bring this feature online, probably via an option specific to the graph data structure API.

Chapter 5

Tacho: Sparse Incomplete Cholesky Factorization

A large sparse system of equations naturally arises from many scienfitic and engineering problems. Often the solution requires sparse matrix factorization *e.g.*, sparse incomplete factorization for preconditioned iterative methods and sparse direct factorization as a subdomain solver in the domain decomposition methods. With a variety of heterogeneous modern computing architectures, it is critical to develop and implement the factorization algorithm in a portable manner. In this work, we demonstrate task parallel approaches for sparse (in)complete Cholesky factorization. Our factorization algorithm uses a 2D block sparse layout and a block represents 2D computing region. Tasks are generated via a right-looking factorization algorithm on the 2D layout and blocks become the basic tasking unit. The spawned tasks are executed asynchronously after their dependences are satisfied. We develop this task parallel sparse factorization, called *Tacho* in `Trilinos/ShyLU` using Kokkos tasking API. Tacho consists of generic task functors that can be executed on Kokkos heterogeneous device abstractions *e.g.*, memory space, execution space and task policy. We successfully port the code to all available Kokkos tasking backends *i.e.*, `P/Q threads`, `OpenMP` and `Cuda`.

5.1 Introduction

Sparse direct methods have been studied for decades in their various forms [25]. While parallel algorithms for complete factorizations have been studied for some time, there are relatively fewer algorithms and implementations for incomplete factorizations. Incomplete Cholesky factorization is effectively used for preconditioned iterative methods to solve large-scale symmetric positive definite (SPD) linear systems. Computing incomplete factorizations scalably in shared-memory systems is challenging because the incomplete factorization is characterized by irregular data access patterns, frequent synchronizations, and loop-carried dependence that limits the available parallelism when expressed in a data parallel manner. First, the incomplete factorizations, by definition, are much more sparse than their counterparts, the complete factorizations. This sparsity precludes the use of any dense linear algebra (DLA) operations such as basic linear algebra subprogram (BLAS) kernels and results in a sparse data access pattern that is very irregular in a traditional incomplete fac-

torization algorithm. By traditional, we refer that the incomplete factorization algorithms are implemented directly using the standard sparse matrix formats *i.e.*, compressed sparse row/column format and rely on `parallel for`, where independent rows (or columns) are factorized and updated in parallel. Inherently, the parallelism is limited by the 1D row (or column) structure of a sparse matrix. Furthermore, the parallel implementation is tightly tied with the algorithm and this may significantly degrade portability of the code.

To remedy all these problems, we propose a task parallel sparse factorization using 2D block sparse layout. Our approach is based on a class of algorithms, called *algorithms-by-blocks*, originating from parallel out-of-core DLA algorithms [26]. This class of algorithms has been adopted for asynchronous thread-parallel execution in DLA libraries [27, 28, 29]. In this approach, a problem can be reformulated by viewing a matrix as a collection of blocks (or tiles); blocks become a computing unit and operations among blocks become tasks. Then, resulting tasks are scheduled, potentially *out-of-order*, to compute resources after satisfying task dependences. This yields a clear separation of concerns by decoupling algebraic structure from runtime task scheduling. Applying this style of algorithms to sparse matrix factorization involves nontrivial challenges in handling the irregular data structure of sparse matrices and blocking strategies that can expose high degree of concurrency.

We have implemented the task parallel Cholesky factorization using portable tasking APIs in Kokkos [30]. Our implementation is available in the ShyLU package [31] of the Trilinos library. Kokkos provides a high-level programming abstractions for modern heterogeneous computing models and harness various tasking backends through unified Kokkos interface. Through the interface, developers write an application code once and the code is portable to heterogeneous device environments with device-specific programming models. The tasking API supports dynamic task DAGs and non-blocking semantics to accommodate devices such as GPUs. Our main contributions of this work include:

- a high-level matrix abstraction for 2D block sparse matrices facilitating task parallelism with dependences using future references;
- a new task parallel implementation for sparse (in)complete Cholesky factorization that utilizes 2D layouts;
- performance evaluation of the new tasking API for several test problems that shows our task-parallel factorization-by-blocks delivers scalable and portable parallel performance.

5.2 Tacho: Task Parallel Sparse Factorization

This section describes task-based level(k) incomplete Cholesky factorization using the 2D sparse block layout.

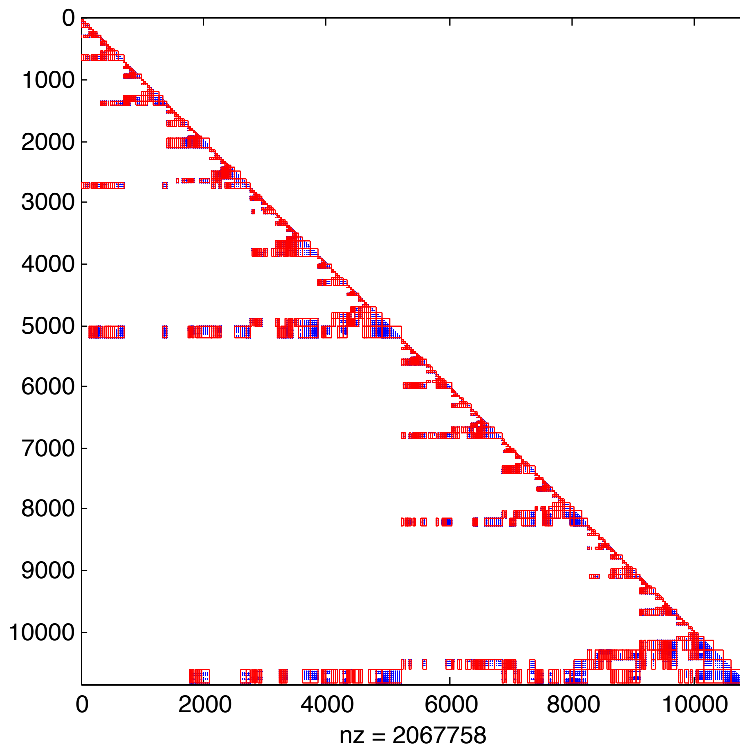


Figure 5.1: 2D Sparse block layout on a matrix (msc10848) selected from the UF collection.

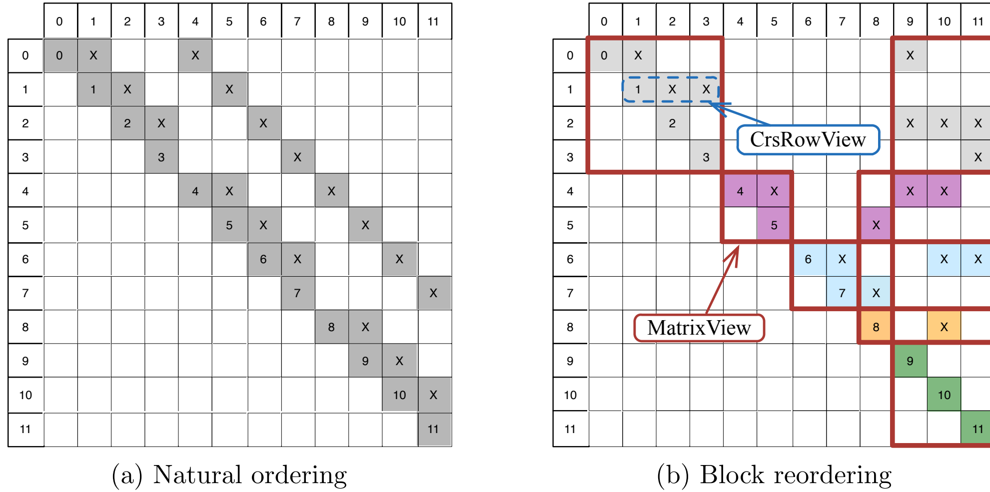
5.2.1 2D Sparse Block Matrix

As the first step of the factorization, we compute nested dissection (ND) [32] ordering using the Scotch [33] library to expose a high degree of concurrency during the factorization. The algorithm recursively separates a sparse matrix into two subproblems, providing a tree hierarchy. After the ordering is applied to to the matrix, symbolic factors[1] are computed using the algorithm proposed by Hysom and Pothen [34].

Our factorization algorithm is uniquely characterized by its recursive usage of the sparse matrix structure; a 2D sparse block layout comprise blocks, each of which defines a computational region on the scalar sparse matrix. An example of 2D sparse block layout is illustrated in Fig. 5.1.

The 2D sparse block layout is identified using Scotch ND hierarchy. More specifically, Scotch provides an array of ranges for columns (or rows) in the reordered matrix where a range corresponds to a group of variables (separator) that can be treated together. Based on the hierarchical relation of those ranges, a 2D sparse block layout is created over the scalar matrix by creating view objects that cover nonzero regions. To facilitate the 2D block layout, we propose the following hierarchy of views on a sparse matrix as illustrated in Fig. 5.2:

[1]the location of potential fill created during numeric factorization



(a) Natural ordering (b) Block reordering

Figure 5.2: A simplified example of symmetric block nested dissection ordering permuted by Scotch. Left: a given sparse matrix with its natural ordering. Right: a 2D sparse block layout on the reordered matrix.

CrsMatrixBase a base matrix object that contains the standard data structure for sparse matrices *i.e.*, row-pointers, column-index array, and value array;

MatrixView a matrix view that defines a 2D rectangular data region overlaid on the base matrix, which is defined by offsets and view dimensions;

CrsRowView a sparse row view that defines the range of columns of a row associated with a `MatrixView`;

We assume that matrices are stored in `CrsMatrixBase` using the standard compressed sparse row (CSR) format. This base matrix has template arguments for a value type which can be either a scalar (for a scalar matrix) or sparse block (for a 2D matrix). A light-weight matrix view is defined as `MatrixView` and includes rectangular partition information such as row/column. The matrix view is templated with an associated base matrix and it may view scalars or blocks within its associated region. A key feature of our task parallel factorization is that this view object becomes a basic computing unit and task granularity is controlled by adjusting the size of a matrix view. For instance, a view can be split into many views or views can be merged into a single view. Since the matrix view only contains meta data, these operations do not carry overhead of data repacking. Our current work does not include precise blocksize tuning capabilities for generating optimal task granularity. Instead, we roughly control the task granularity by adjusting the Scotch tree hierarchy level. To access entries in a matrix view, the The `CrsRowView` is used to access elements in a matrix view.

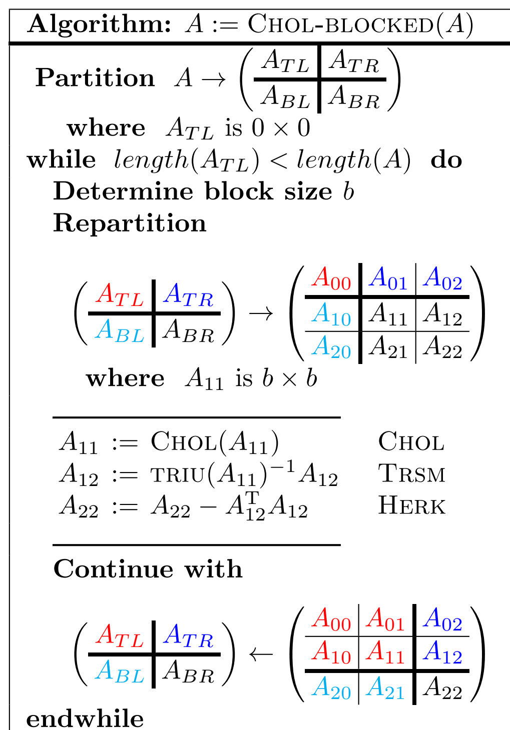


Figure 5.3: Cholesky algorithm. The blocks in the 2×2 and 3×3 block matrices that correspond to each other are of the same color. CHOL and TRIU represent Cholesky factorization and the upper triangular part of an input matrix respectively.

5.2.2 Cholesky-by-blocks

The right-looking Cholesky algorithm is shown in Fig. 5.3. The algorithm is expressed with partitioned matrices using Formal Linear Algebra Methods Environment (FLAME) notations [35, 36]. A short description of the notation follows. First, note that the algorithm will work equally well on matrices with scalar entries (b=1) or sparse block entries (variable b). Second, A_{BR} is partitioned further and updated at each iteration of the `while` loop. In this particular algorithm, note that the computations only happen on the A_{BR} block. The algorithm consists of three different operations. Using the BLAS notation, the three operations in Fig. 5.3 are CHOL, TRSM, and HERK which correspond to a Cholesky factorization, triangular solve and hermitian rank-k update. Finally, the partition is redefined (see the thick partition line moving forward) for the next iteration of the algorithm.

We transform this algorithm into so-called Cholesky-by-blocks by converting the basic computing unit from a scalar to a block. For a given 2D sparse block layout, we apply the Cholesky algorithm described in Fig. 5.3 to $\boldsymbol{A}_{ij}$ elementwise as described in Fig. 5.4. By doing so, the three different operations inside the while loop becomes three different opportunities to generate tasks. As a result, our task-parallel Cholesky factorization has the same look-and-feel as the scalar Cholesky factorization, greatly improving programmability.

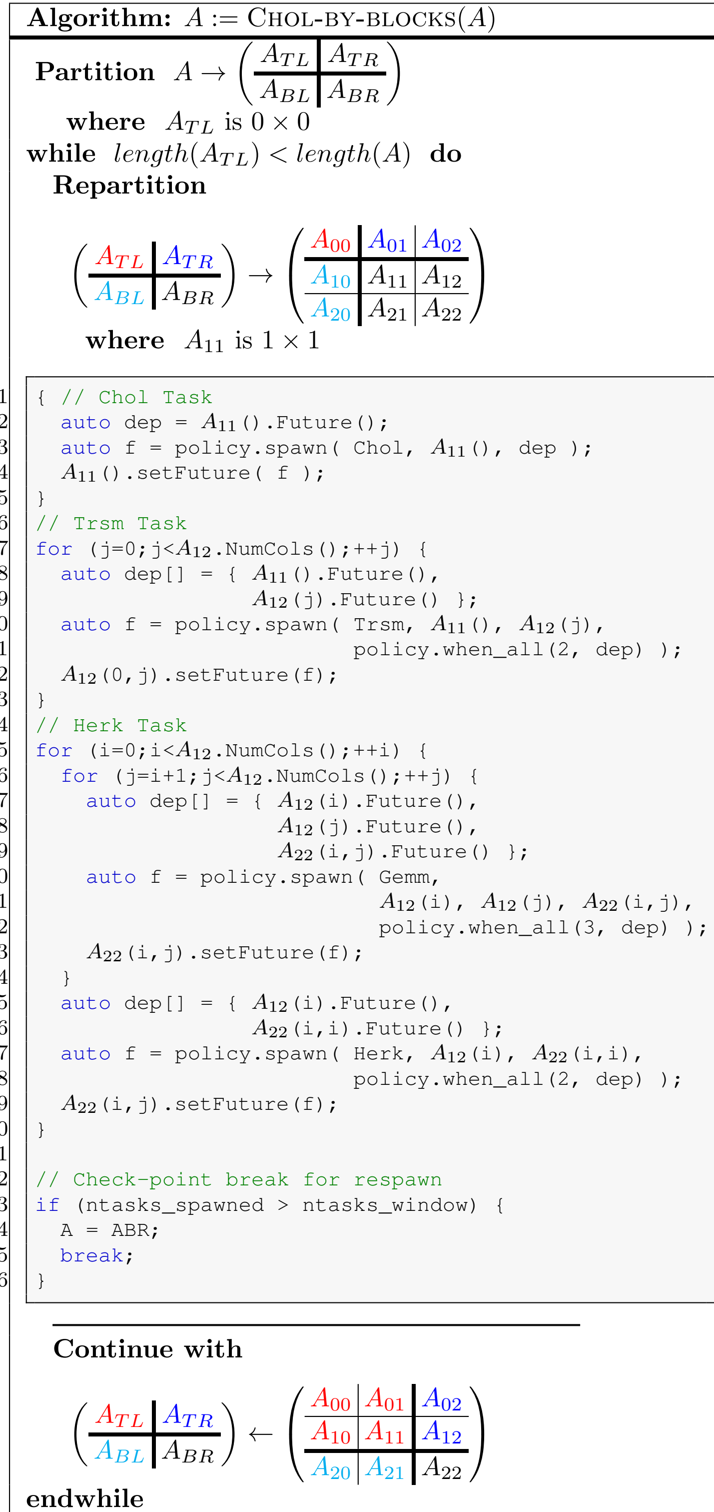


Figure 5.4: Cholesky-by-blocks algorithm on a 2D block layout.

The Cholesky-by-blocks algorithm creates tasks and spawned with dependences. A

spawned task updates the *future* of an output block associated with the task. Since a block records associated tasks as a chain of futures, we do not need to keep track of the entire task dependences but only follow the loop body of the algorithm.

We demonstrate this algorithm with a small example matrix. Suppose that ND ordering provides a symmetric permutation matrix $\boldsymbol{P}$, which leads to an upper triangular block matrix

$$\boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P} = \begin{pmatrix} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ & & & & A_{44} \end{pmatrix}$$

where all $\boldsymbol{A}_{ij}$ blocks are sparse and have block dimensions compatible with each other. Then, we apply the Cholesky-by-blocks algorithm as depicted in Fig. 5.4. As a result, a sequence of block matrix computations is generated as illustrated in Fig. 5.5.

Task dependences are found from the input/output relations described in the loop body of the Cholesky algorithm:

$$\text{Chol} \rightarrow \text{Trsm} \rightarrow \text{Herk (or Gemm)}.$$

For example, a Trsm task will depend on a Chol task that is an input for the task; a Herk task will depend on Trsm tasks that should be completed on the data region that is required as an input of the task. This dependence relationship is applied to individual blocks and a corresponding task DAG is shown in Fig. 5.6b. Contrary to the coarse grain tasks that correspond to the block ND tree depicted in Fig. 5.6a, our Cholesky-by-blocks generates a larger number of asynchronous fine-grained tasks. In this particular example, the Chol(A_{22}) in the third iteration can be executed before finishing tasks (*i.e.*, Trsm, Herk and Gemm) created in the first and second iterations. This is possible because the Chol(A_{22}) has input dependences only to itself. Again, we note that the Cholesky-by-blocks algorithm does not require separate dependence analysis; instead, it iterates the `while` loop on the entire 2D matrix and generates tasks as it steps through the loop. Tasks such as Chol(A_{22}) can be run immediately as they are created with no dependences. This is possible because our task parallel approach is not strictly tied to the tree hierarchy derived from ND ordering. Furthermore, we see in the first iteration that Trsm(A_{04}) can begin immediately as its dependences are satisfied, whereas Chol(A_{22}) won't be even created till the second iteration, which is the opposite of what a tree based algorithm would have done. This approach exposes much more fine-grained task parallelism. Also note that Herk(A_{33}) in the third iteration has taken the form of a sparse Gemm which is much more cache-friendly to compute than simple rank-1 updates.

Difference from other approaches After we construct a 2D matrix based on ND block ordering, we no longer use the tree hierarchy to extract parallelism. Tasks are created by the Cholesky-by-blocks based on the 2D block layout.

$$\left(\begin{array}{c|cccc} A_{00} & & & & A_{04} \\ \hline & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ & & & & A_{44} \end{array}\right) \qquad \begin{array}{lcl} A_{00} & := & \textsc{Chol}(A_{00}) \\ A_{04} & := & \textsc{triu}(A_{00})^{-1}A_{04} \\ A_{44} & := & A_{44} - A_{04}^T A_{04} \end{array}$$

(a) 1st iteration

$$\left(\begin{array}{c|c|ccc} A_{00} & & & & A_{04} \\ \hline & A_{11} & & A_{13} & A_{14} \\ \hline & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ & & & & A_{44} \end{array}\right) \qquad \begin{array}{lcl} A_{11} & := & \textsc{Chol}(A_{11}) \\ A_{13} & := & \textsc{triu}(A_{11})^{-1}A_{13} \\ A_{14} & := & \textsc{triu}(A_{11})^{-1}A_{14} \\ A_{33} & := & A_{33} - A_{13}^T A_{13} \\ A_{34} & := & A_{34} - A_{13}^T A_{14} \\ A_{44} & := & A_{44} - A_{14}^T A_{14} \end{array}$$

(b) 2nd iteration

$$\left(\begin{array}{cc|c|cc} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ \hline & & A_{22} & A_{23} & A_{24} \\ \hline & & & A_{33} & A_{34} \\ & & & & A_{44} \end{array}\right) \qquad \begin{array}{lcl} A_{22} & := & \textsc{Chol}(A_{22}) \\ A_{23} & := & \textsc{triu}(A_{22})^{-1}A_{23} \\ A_{24} & := & \textsc{triu}(A_{22})^{-1}A_{24} \\ A_{33} & := & A_{33} - A_{23}^T A_{23} \\ A_{34} & := & A_{34} - A_{23}^T A_{24} \\ A_{44} & := & A_{44} - A_{24}^T A_{24} \end{array}$$

(c) 3rd iteration

$$\left(\begin{array}{ccc|c|c} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ \hline & & & A_{33} & A_{34} \\ \hline & & & & A_{44} \end{array}\right) \qquad \begin{array}{lcl} A_{33} & := & \textsc{Chol}(A_{33}) \\ A_{34} & := & \textsc{triu}(A_{33})^{-1}A_{34} \\ A_{44} & := & A_{44} - A_{34}^T A_{34} \end{array}$$

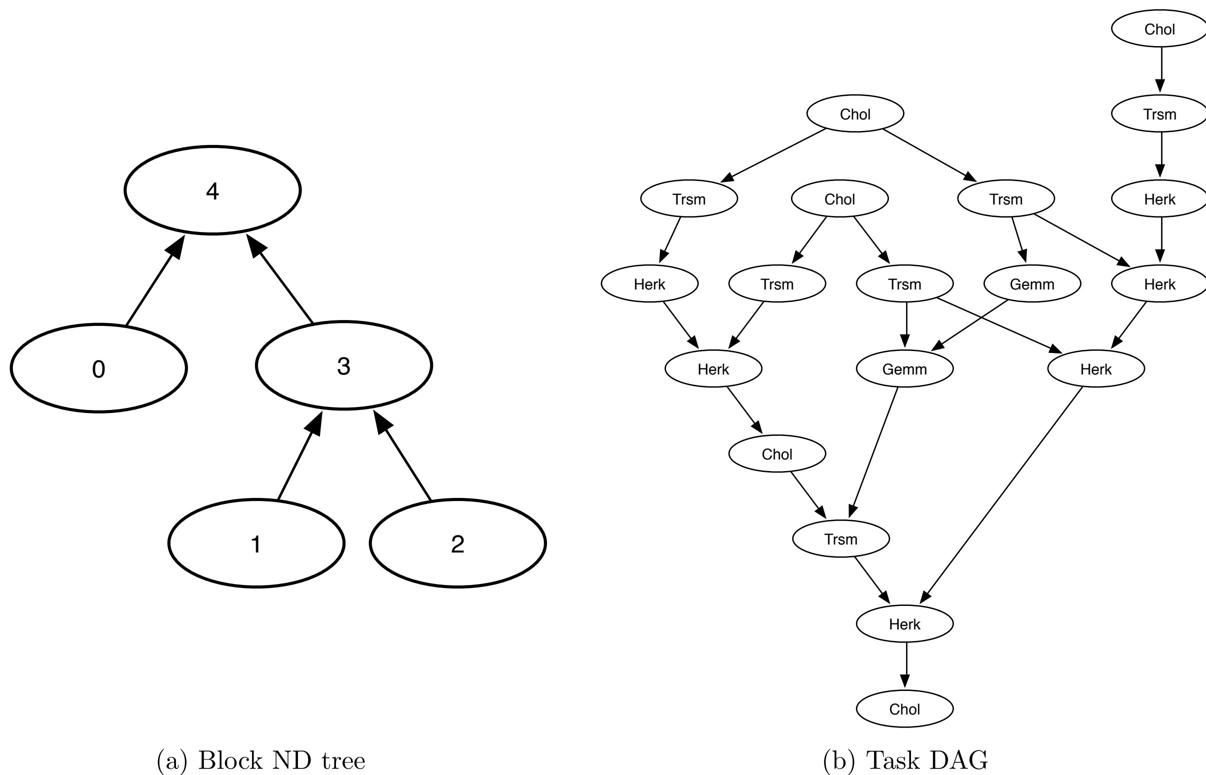
(d) 4th iteration

$$\left(\begin{array}{cccc|c|} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ \hline & & & & A_{44} \\ \hline \end{array}\right) \qquad \begin{array}{lcl} A_{44} & := & \textsc{Chol}(A_{44}) \end{array}$$

(e) 5th iteration

Figure 5.5: Generated block matrix computations while proceeding on Cholesky-by-blocks.

The approach differs from others in that we do not explicitly rely on the ND tree hierarchy (or the elimination tree) in the numeric factorization. On the other hand, conventional approaches for task parallel implementation explicitly use the tree hierarchy to generate independent tasks and their dependences, as well as to distribute compute resources according to subtree structures [37]. Such implementations may not be performance-portable as the implementation is hard-wired to problem specific sparse structures and hardware execution



(a) Block ND tree (b) Task DAG

Figure 5.6: A task dependence graph for the example illustrated in Fig. 5.5.

environments.

5.2.3 Implementation Details using Kokkos Task API

```
1 int main() {
2   Kokkos::TaskPolicy<DeviceSpaceType> policy( memory_pool_size );
3
4   const auto future = policy.host_spawn( CholByBlocks( policy, A ) );
5
6   Kokkos::wait( policy );
7
8   return future.get();
9 }
```

Figure 5.7: Main driver for task spawning from the host execution space.

Tacho consists of generic task functors to compute block matrix operations and to generate tasks. Beginning with the `main` driver routine, Fig. 5.7 shows how a task policy is created and a task is spawned to generate tasks, where A represent the 2D sparse block layout (matrix of matrices). This top-level task generator is always created and spawned

```
template<typename PolicyType,
         typename MatrixType>
class CholByBlocks {
public:
  typedef typename PolicyType::member_type member_type;
  typedef int value_type;

private:
  PolicyType policy;
  MatrixType A;

public:
  // Constructor
  CholByBlocks(const PolicyType &policy, const MatrixType &A);

  // Task functor
  KOKKOS_INLINE_FUNCTION
  void operator()(member_type &member, value_type &r_val) {

    // only team leader can generate tasks
    if (member.team_rank() == 0) {
      // Cholesky-by-blocks periodically returns to manage
      // the memory foot prints during task execution and
      // A is updated with a remained block, ABR
      const int ierr = generate_tasks_for_chol_by_blocks(A);

      // check if factorization is completed
      if (A.NumRows()) {
        // this task is respawned with a low priority
        // and task generation is suspended until other
        // high priority tasks in the queue are processed
        policy.respawn(this, Kokkos::TaskLowPriority);
      } else {
        // nullify the future recorded in this matrix
        // for fast retiring of the referenced task
        A.setFuture(typename MatrixType::future_type());
      }
      r_val = ierr;
    }
  }
}
```

Figure 5.8: A task to generate tasks for Cholesky-by-blocks.

via `host_spawn`. This spawned task is immediately executed and it starts generating tasks using `task_spawn`. An example functor for Cholesky-by-blocks factorization is shown in Fig. 5.8 and the details of the `generate_tasks_for_chol_by_blocks` routine is illustrated in Fig. 5.4. Note that task generators can recursively spawn other task generators with dependences and tasks are dynamically spawned and processed in parallel before all tasks are generated. In this particular example, the task generator periodically stops and it respawns itself to complete factorization. In this way, we can control the number of active tasks in the queue and this constraint is required to solve different sizes of problems using a fixed size of memory pool. It also helps manycore processors with slow clock speed as it overlap task generations and processing actual computations.

Matrix ID	# of rows(n)	# of nonzeros(nnz)	nnz/n
ecology2	999,999	4,995,991	4.99
G3_circuit	1,585,478	7,660,826	4.83
thermal2	1,228,045	8,580,313	6.98
pwtk	217,918	11,524,432	52.88

Table 5.1: Test problems selected from the UF sparse matrix collection.

5.3 Performance Evaluation

We evaluate our task parallel incomplete Cholesky factorization on problems selected from the University of Florida sparse matrix collection [38]. The matrix properties are tabulated in Tab. 5.1. The largest problem, `G3_circuit`, has about 1.5 million rows. Note that `pwtk` is relatively denser by an order of magnitude than other test problems (see the average number of non-zeros per row in the table). Later, we show that this property significantly changes both serial and parallel performance. We use three different multicore and manycore platforms for the evaluation:

1. a dual socket, "Sandy Bridge" processors, 2×8 Intel Xeon E5-2670 2.6GHz cores;
2. a "Knights Corner" coprocessor, 1×57 Intel Xeon Phi with 1.1GHz cores;
3. a dual socket, IBM POWER8 system, 2×10 3.4GHz cores and 5 cores per NUMA region.

Three levels of cache system is used in the Sandy Bridge processors *i.e.*, a large shared 20MB L3 cache and a private 256KB L2 cache. The Intel Xeon Phi and IBM POWER8 processors are working with 4 hyperthreads and 8 hardware threads respectively. Each socket in the IBM POWER8 system has two non-uniform memory access (NUMA) regions with its own memory controller. This results in higher latency between processors within a socket and between sockets than local accesses. In each NUMA region, there exist three levels of cache with a private 512 KB L2 and the L3 being a collection of shared 58MB segments. In our performance evaluation, a single thread is mapped to a single core and we do not use hyperthreading in Intel architectures nor symmetric multithreading in the IBM processor.

5.3.1 Task Parallel Cholesky-by-blocks Factorization

First, we perform the symbolic factorization to determine the location of fill for the level(k) incomplete Cholesky factorization. Similar to Hysom and Pothen [34], our symbolic factorization performs breath first search (BFS) on the adjacency graph of a matrix in parallel for each node to specify level(k) fill structure. This is implemented in a scalable fashion

	# of nonzeros in U [millions]				
Matrix ID	L0	L1	L2	L4	Chol(AMD)
`ecology2`	2.9	4.7	6.0	8.3	45.7
`G3_circuit`	4.6	7.4	9.6	14.5	189.1
`thermal2`	4.9	7.9	10.6	14.8	64.8
`pwtk`	5.9	10.9	15.0	21.4	60.0

Table 5.2: Number of nonzero Cholesky factors resulting from level(k) symbolic factorization; for comparison, the last column shows the number of fill from complete factorization with AMD ordering.

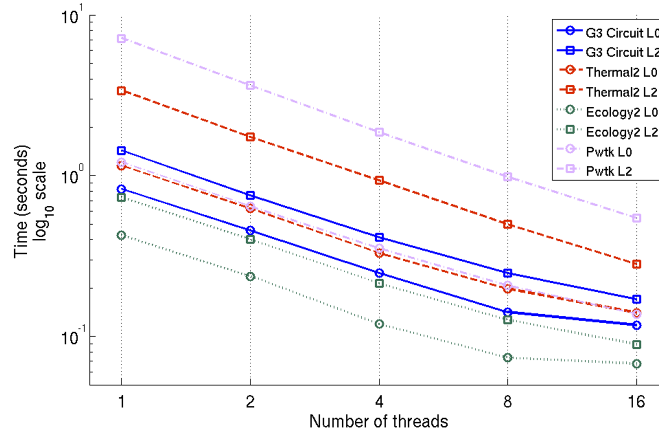
using two Kokkos parallel patterns: `parallel_for` and `parallel_scan`. The numbers of nonzero Cholesky factors generated by the level(k) incomplete Cholesky factorization are summarized in Tab. 5.2. All matrices are reordered with the block ND algorithm provided by the Scotch library [33]. For comparison, we also provide size of the fill for complete Cholesky factorization in the last column of the table.

Parallel performance We report strong scalability of our task parallel level(k) incomplete Cholesky factorization in Fig. 5.9. One can see that our task parallel Cholesky-by-blocks using 2D sparse block layout performs scalably for all testbeds. The parallelism induced from irregular sparse matrices and different fill patterns could vary significantly. However, the use of 2D block matrices and task parallel approach shows robust parallel performance across different test problems. For comparison with other solver packages, we refer our other publications [39, 40].

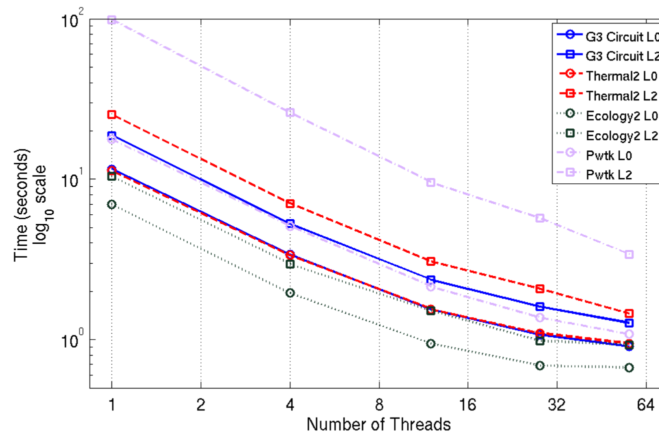
Task granularity and overhead Finding appropriate task granularity is very important to attain higher parallel performance. Multiple aspects of performance trade-offs should be considered to determine optimal task granularity:

- total number of generated tasks,
- level of concurrency expressed from sparse factorization,
- tasking overhead (context switching, task creation, scheduling and destruction),
- data access overhead for multiple sparse kernel launching,
- number of computing units and local cache sizes.

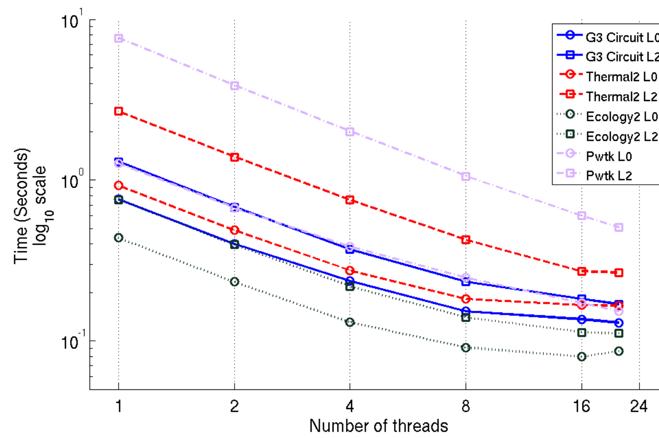
Using many fine-grained tasks results in a higher degree of concurrency which is more suitable for manycore computing environments. However, using such a large number of tasks may significantly increase tasking overhead and irregular data access cost, which decreases overall



(a) Sandy Bridge



(b) Intel Xeon Phi



(c) IBM POWER8

Figure 5.9: Time for level(k) incomplete Cholesky-by-blocks factorization on testbeds: Sandy Bridge, Intel Xeon Phi and IBM POWER8. The time complexity includes both symbolic and numeric factorization.

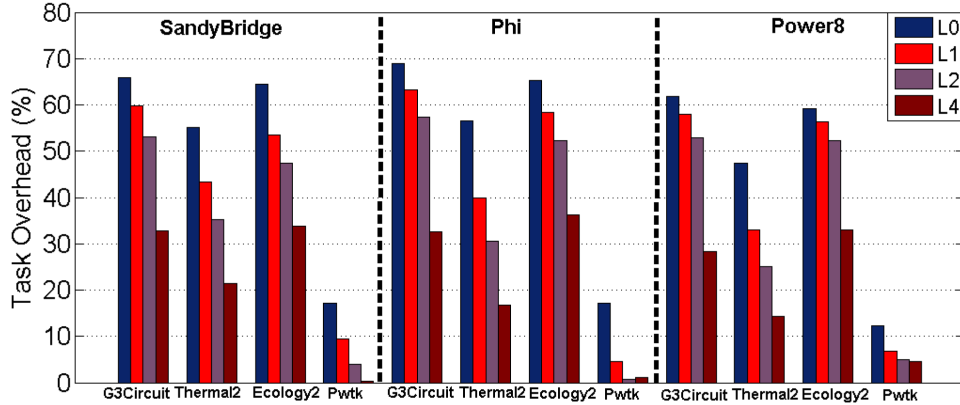


Figure 5.10: Tasking overhead for different matrices with different level of fills.

parallel performance of sparse factorization. On the other hand, generating coarse grained tasks can decrease tasking overhead but may not expose enough concurrency to use all available hardware resources.

To explore this performance trade-off, we plot the relative tasking overhead, $1 - T_{\text{serial}}/T$, where T is the time cost of task parallel Cholesky-by-blocks using a single thread and T_{serial} is the time cost of serial sparse Cholesky factorization. Fig. 5.10 describes the relative tasking overhead on the Sandy Bridge, the Xeon Phi and the IBM POWER8 processors. Our task parallel approach may include two different types of overhead: 1) task scheduling overhead that increases proportionally with the number of generated tasks and 2) overhead due to irregular data access during the asynchronous task execution. From the figures, `pwtk`, which is relatively less sparse compared to others, shows different performance trends from the others. For convenience, we compare `ecology2` to represent the other sparse matrices. Some key observations are:

- `ecology2` exhibits higher tasking overhead than `pwtk` due to its lower computational workload in each task (for the level 0 factorization, `ecology2` matrix carries almost the same amount of overhead as the numerical factorization while the tasking overhead in `pwtk` is almost negligible);

- with an increasing level of fill, relative tasking overhead of both test problems decreases as the workload associated with each task increases.

The other test problems demonstrate similar performance behaviors to these two representative cases.

5.4 Summary

In this chapter, we have presented an algorithm for task-parallel incomplete Cholesky factorization that applies *algorithms-by-blocks* factorization to a 2D block sparse matrix. We have shown that by encoding the ND tree hierarchy in the 2D block matrix, the task DAG need not be restricted to a simple ND tree. This results in a much richer task DAG, leading to better performance. We believe this algorithm opens up a new direction of research in which other sparse factorizations such as LU and QR could also gain performance benefits by following the similar pattern used here.

With Kokkos tasking API, we have implemented a generic version of task parallel sparse Cholesky factorization and demonstrated the portable performance of the factorization across different types of computing architectures. The effective performance of our factorization largely depends on the task granularity. Finding an optimal task granularity in sparse matrix factorization is very difficult as it depends on various problem and architecture specific tuning parameters such as the number of generated tasks, ratio between workload and tasking overhead and other factors. Our performance evaluation shows that such tasking overhead can be effectively amortized when tasks are compute-intensive. As future work, we plan to extend this approach for supernodal direct factorization exploiting BLAS level 3 operations.

Chapter 6

miniTri: A Data Analytics Mini-Application

The Graph BLAS effort to standardize a set of graph algorithms building blocks in term of linear algebra primitives promises to deliver high performing graph algorithms and greatly impact the analysis of big data. However, there are challenges with this approach, which our data analytics miniapp miniTri exposes. In this section, we describe a task-parallel approach to linear algebra-based miniTri formulation, addressing these challenges and describing a Kokkos/Qthreads task-parallel implementation that performs as well or slightly better than the highly optimized, baseline OpenMP data-parallel implementation. Most of this work is published here [41].

6.1 Background

6.1.1 Linear Algebra Primitives for Graph Algorithms

Motivated by the success of the standardization of dense linear algebra primitives in BLAS [42] and LAPACK [43] and their impact on computational science and engineering (CS&E) applications, there have been recent efforts to produce a standard set of graph algorithm building blocks for graph analysis and related data science applications. One such effort, Graph BLAS [44], attempts to standardize the building blocks of graph algorithms in the language of linear algebra. This effort exploits the duality between graphs and matrices, which has already been highly impactful in CS&E algorithms (e.g., data partitioning, solvers), to develop overloaded linear algebra-based operations that express fundamental graph algorithms. Previous work has shown that these overloaded linear algebra kernels can express many useful graph computations [45], including breadth-first search, betweenness centrality, and triangle counting [46]. There are several concrete implementations related to the Graph BLAS standardization effort: CombBLAS [47], D4M [48], Graphulo [49], and GraphMat [50].

Important features of Graph BLAS include the conciseness of the representation and the promise of high performance. The Graph BLAS primitives use a high level of abstraction to express complicated graph algorithms in just a few overloaded linear algebra operations, which each typically represent many fine-grain operations on the graph. As for performance,

if hardware vendors could leverage previous experience in developing sparse linear algebra libraries to effectively optimize a small number of generalized sparse linear algebra operations, this Graph BLAS approach would result in highly optimized graph analysis applications.

This work, which focuses on a graph analytics miniapplication miniTri that represents classes of strange, unexpected graph computations [51], is part of this effort to express more complex analytics with Graph BLAS inspired linear algebra primitives. We believe that miniTri, described more fully in the next subsection, is an important stressor of the Graph BLAS standard, requiring flexibility and asynchronous execution of its fine grain operations (underlying its high-level abstraction) in order to achieve high performance and scalability. An efficient Graph BLAS implementation of miniTri would go a long way to validating the Graph BLAS approach.

In this work, we focus on a specific task-parallel approach to the Graph BLAS inspired miniTri formulation. Our key contributions are:

- We discuss a task-parallel approach to miniTri and how it addresses challenges intrinsic to Graph BLAS,
- We introduce the Kokkos task-parallel API and describe how to implement this linear algebra-based miniTri efficiently in a Kokkos/Qthreads task-parallel environment, and
- We provide strong computational evidence that this task-parallel approach exhibits similar or better performance to a more traditional task-parallel approach.

6.1.2 miniTri: A Data Analytics Miniapp

Proxy applications or miniapps, standalone applications that represent key sets of kernels of a larger application, are important for ensuring that real applications achieve realized high performance on emerging high performance computing (HPC) architectures. In previous work, we introduced a new miniapp miniTri, developed at Sandia National Laboratories, to address the paucity of data analytics miniapps [51]. miniTri, which is a part of the Mantevo suite of miniapps [52], differs from standard HPC data analytics miniapps/benchmarks found in SSCA-2 [53] and Graph 500 [54] in that it is not based on neighbor set expansion. The primary kernel in miniTri is triangle enumeration, which offers different challenges than traditional graph search [51] and is critical for several data science problems, including dense subgraph detection (with applications to cyber security, intelligence, and functional biology) [55, 56].

In a nutshell, miniTri applies several operations to an input graph in order to find an upper bound on the largest clique in that graph. First, given an input graph, all triangles in that graph are enumerated or listed. Then, the triangle vertex degree t_v (number of triangles incident on vertex v) is computed for each vertex v in the graph; and the triangle edge degree t_e (number of triangles incident on edge e) is computed for each edge e in the graph. Finally,

a value k is computed for each triangle t in the graph using the following formula:

$$\arg\max_k\{(\min_{v\in t} t_v \geq \binom{k-1}{2}) \cap (\min_{e\in t} t_e \geq k-2)\}.$$

The output of miniTri is a histogram that summarizes the number of triangles that corresponds to each value of k.

There are several different ways to implement miniTri. In this work, we focus on the previously developed Graph BLAS like approach that expresses miniTri concisely in terms of linear algebra primitives (for more details on this approach including a worked example see [51]). Fig. 6.1 shows this basic linear algebra approach, where miniTri can be expressed in terms of three overloaded linear algebra operations and one function with linear algebra structure inputs. This linear algebra based implementation takes the adjacency matrix A and the incidence matrix B of the graph as inputs. In the first step (line 3), an overloaded sparse matrix-matrix multiplication of the adjacency and incidence matrices is used to enumerate all triangles in the graph. This step enumerates each triangle three times ($C = L \cdot B$, where L is the lower triangular part of A would enumerate each triangle once), which although inefficient, makes the triangle degree calculations simpler. By construction, each triangle is represented once in C for each of its edges and vertices. Thus, the triangle vertex and edge degrees are simply the number of nonzeros in the rows and columns, respectively. The overloaded sparse matrix-dense vector multiplication operations in lines 4 and 5 count the number of nonzeros in the rows and columns, thus yielding the triangle vertex and edge degrees. In final step, we compute k for each triangle using above formula and the linear algebra data structures C, t_v, and t_e, and form the desired histogram. In this manner, we can write miniTri in terms of this very simple set of linear algebra operations, which can be implemented using standard linear algebra data structures with integer elements.

```
1: procedure miniTri(A, B)
2:     k_cnts ← 0
3:     C = A · B                              ▷ Enumerate triangles
4:     t_v = C · 1                            ▷ Calculate triangle vertex degree
5:     t_e = C^T · 1                          ▷ Calculate triangle edge degree
6:     k_cnts = kcount(C, t_v, t_e)           ▷ Compute k for triangles
7: end procedure
```

Figure 6.1: Linear Algebra Based Formulation of miniTri.

6.2 Task-Parallel miniTri Overview

6.2.1 Challenges with Linear Algebra Based miniTri

The linear algebra based miniTri formulation as described in subsection 6.1.2 is appealing in that miniTri can be written in a very concise form, in terms of a few linear algebra operations.

However, there are at least two challenges with this approach, especially when it comes to developing a parallel implementation.

First, there is a performance challenge with this linear algebra based approach. Modern processor architectures, with their numerous cores, require applications to be able to express sufficient parallelism in order to fully utilize these computational resources. To fully utilize these resources, load balancing of the computational work becomes extremely important. In bulk synchronous parallel models, even small discrepancies in the run time between concurrently executing threads can lead to significant idle time when summed across all the cores and poor parallel performance. Load balancing for these graph analysis applications is particularly challenging, in general, due to the irregular nature of the computations and the varied memory access times (due to poor memory locality). However, the use of bulk synchronous methods (traditionally used in these linear algebra based approaches) acerbates this problem and limits performance due to inherent synchronization bottlenecks between each of the kernel calls. Although with a multithreaded data-parallel approach, work stealing can help with the load imbalance within each kernel, without absolute perfect load balance (a virtual impossibility for complicated data science problems running on dynamic systems), cores will become imbalanced and there will be associated idle time at the end of each kernel.

The second, perhaps more limiting challenge for this linear algebra based approach to miniTri (if implemented in this traditional bulk synchronous parallel manner) is that it requires all the triangles in C to be stored. This is not a requirement of miniTri itself, however, where the triangles are simply intermediate results used to compute triangle degrees and resulting k values, but one imposed by the artificial global synchronization points between the kernels. Having to store all triangles would severely limit the size of the solvable problems, since there are in a worst case $O(E^{3/2})$ triangles (where E is the number of edges) in the graph, and it is not unusual to have 100-1000 triangles per edge in graphs.

6.2.2 Task-Parallel Linear Algebra Based miniTri

In this work, we focus primarily on the load-imbalance challenge although we lay the ground work for a solution to the memory challenge as well, which we plan to address once we have run-time support for memory-constrained scheduling. Our goal is to introduce more asynchrony into this Graph BLAS inspired miniTri implementation and obtain similar performance to highly optimized data-parallel implementations. In future work, this asynchrony will be further exploited to address this memory challenge as well.

We advocate using a task-parallel approach to address these challenges as illustrated in Fig. 6.2. With this task-parallel approach, the problem is typically over decomposed into many more tasks than there are computational cores on the target architecture. Each task is assigned a subset or block of the operations that make up a linear algebra kernel (e.g., operations on a block of matrix rows or operations on a 2D block of matrix nonzeros). All explicit barriers between kernels are removed (with the exception of truly blocking operations such as print functions that require global consistency of memory). These barriers

are replaced by fine-grain dependences between tasks. A key feature in this approach is asynchrony, which allows tasks in subsequent kernels to make progress before all the tasks in the first kernel finishes, allowing the scheduler to decrease the idle time between kernels. Along these lines, we have implemented two task-parallel versions of the linear algebra based miniTri: one Kokkos [2] and Qthreads [1] based implementation and one HPX [57] based implementation. While the HPX implementation supports distributed memory task parallelism, the focus of this work is shared-memory (single node) task parallelism, which is provided by the Kokkos/Qthreads implementation.

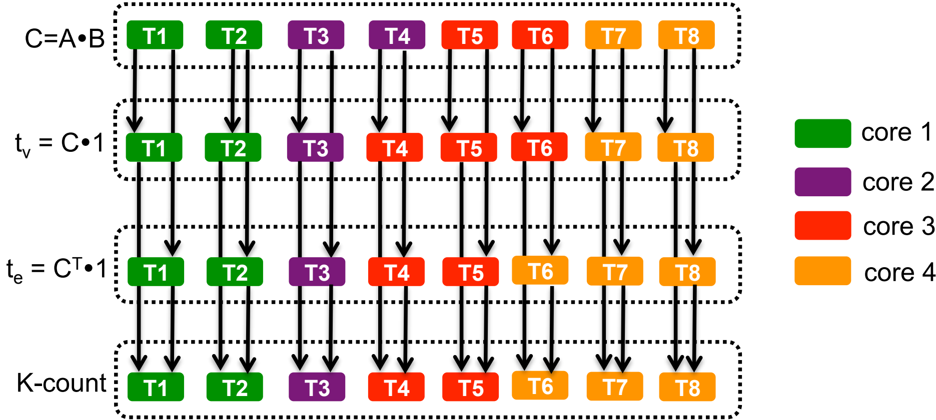


Figure 6.2: Illustrative example of task-parallel approach to linear algebra based miniTri. Arrows represent dependencies between tasks.

6.3 Task-Parallel Kokkos/Qthreads miniTri Implementation

6.3.1 General Implementation Details

The task-based linear algebra formulation of miniTri described above was implemented using the Kokkos programming model. For the matrices in this implementation, a compressed sparse row (CSR) matrix format was used. Kokkos was used to manage the dynamic memory of the CSR matrix and the vector structures. For this task-parallel implementation, the linear algebra work was divided among the tasks using a 1D block distribution, such that each task executed a set of operations that corresponds to a block of rows (with the block size being defined by a parameter). Kokkos atomics were used to increment the shared data structures in a thread safe manner.

6.3.2 Kokkos Task-Parallel Functionality

In this miniTri implementation, task-parallel operations in the Kokkos interface are controlled through the Kokkos Task Policy data structure, which is templated upon the execution space (Qthreads for this work). In developing the miniTri task-parallel software, the four primary computational kernels (lines 3-6 in Fig. 6.1) were broken down into many tasks, where the number of tasks is typically much larger than the number of cores on the architecture. Each kernel is decomposed into tasks using the following three steps:

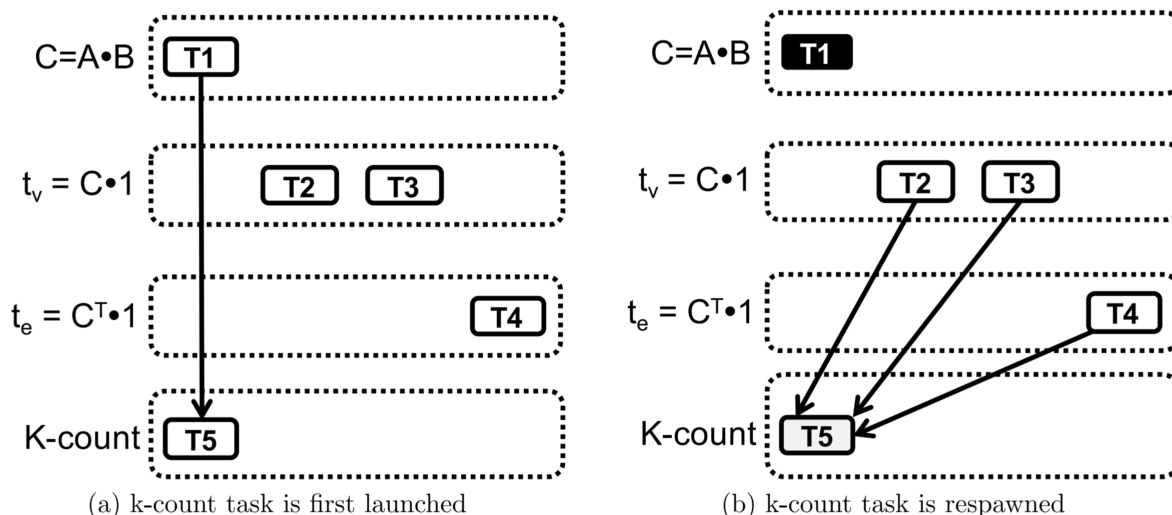
For the decomposition of the key kernels into tasks, a "fire-and-forget" paradigm is used, where each kernel returns immediately after its tasks are launched, although the corresponding tasks have most likely not finished executing. This pattern produces a truly asynchronous task-parallel data flow with few global synchronization points. Global synchronization points are only added for functions such as printing global values where consistency is needed across all the data. Such calls should be avoided when possible.

6.3.3 Dynamic Task Dependencies of miniTri

miniTri, however, has particularly complicated dependences, especially for the k-count tasks. The challenge for the k-count tasks is that with the purely asynchronous ("fire-and-forget") approach, there is no guarantee that all the dependencies of the k-count tasks are known when the k-count tasks are created. At task creation, each k-count task knows which single triangle enumeration task it depends upon. However, until the corresponding triangle enumeration task is complete, the triangles are not known and thus the k-count task does not know what triangle edge and vertex degree tasks that it depends upon. Fortunately, Kokkos provides a *respawn* function that facilitates a solution to this complication, using the following procedure. After the task creation, only the dependence of the k-count task on the triangle enumeration task is set. When the k-count task is scheduled for execution, the triangle enumeration task on which the k-count task depends has completed and thus the dependencies on the triangle vertex and edge degree tasks are known. At this point, the dependencies for the k-count task are reset to these triangle vertex and edge degree tasks (removing the triangle enumeration task dependency, which is no longer necessary)and the Kokkos respawn function is called to relaunch the task. Once the task is relaunched, the task can be rescheduled for execution after the triangle degree task dependencies are satisfied.

To illustrate this more concretely, Figs. 6.3a and 6.3b demonstrate this pattern with a very simple example. When k-count task T5 is initially spawned only the dependency on triangle enumeration task T1 is known (Fig. 6.3a). When k-count task T5 executes for the first time, the triangle enumeration task T1 has aleady completed. At this point, it is known that k-count task T5 should also depend on triangle vertex degree tasks T2 and T3 and triangle edge degree T4. Thus, k-count task T5 sets these dependencies and respawns itself (Fig. 6.3b). When the k-count task T5 is rescheduled for execution all of its dependencies have been met, and it can run to completion. This fairly unobtrusive respawn technique allows these complicated, dynamically calculated dependencies to be addressed

while maintaining the desired asynchronous task parallelism.



(a) k-count task is first launched

(b) k-count task is respawned

Figure 6.3: Illustration of complex dependences for k-count tasks in miniTri.

6.4 Results

In this section, we present numerical experiments for the Kokkos/Qthreads task-parallel implementation of the linear algebra based miniTri and compare the results to a baseline OpenMP data-parallel implementation of the linear algebra based miniTri. These results were conducted on a 16 core workstation with 64 GB of memory (dual processor Intel Xeon E5-2630 v3 @ 2.40 GHz). Both implementations were compiled using gcc version 4.8.3.

Table 6.1 shows a summary table for the datasets analyzed for our numerical experiments. These datasets were chosen to be representative of different size data science problems that could fit into the main memory of the workstation (i.e., a relatively small number of triangles and a low ratio of the number of triangles to the number of edges). The Oregon-1 (autonomous system dataset) and the com-Youtube (social network dataset) graphs were obtained from the SNAP collection [58]. The email-Enron (Enron email network) and the ca-AstroPh (Arxiv Astro Physics collaboration network) were also part of the SNAP collection but were obtained from the University of Florida Matrix Collection [59].

Fig. 6.4 shows the Kokkos/Qthreads miniTri implementation execution times for the Oregon-1 graph and different block sizes (the number of rows on which each task operates) as we increase the number of threads that will be executing the tasks. The "sweet spot" seems to have tasks operate on 100 rows. A block size of 10 rows does not have enough work per task to overcome the runtime system overhead; block sizes of 1000 or more do not have enough parallelism to fully utilize the machine since there are only 11k vertices in this graph. We see execution time improvements up to 16 threads, for block sizes of 10 and 100.

Table 6.1: Datasets

Matrix	$\|V\|$	$\|E\|$	$\|T\|$	$\|T\|/\|E\|$
Oregon-1	11174	23409	19894	0.85
email-Enron	36692	183831	727044	3.95
ca-AstroPh	18772	198110	1352469	6.87
com-Youtube	1157827	2987624	3056386	1.02

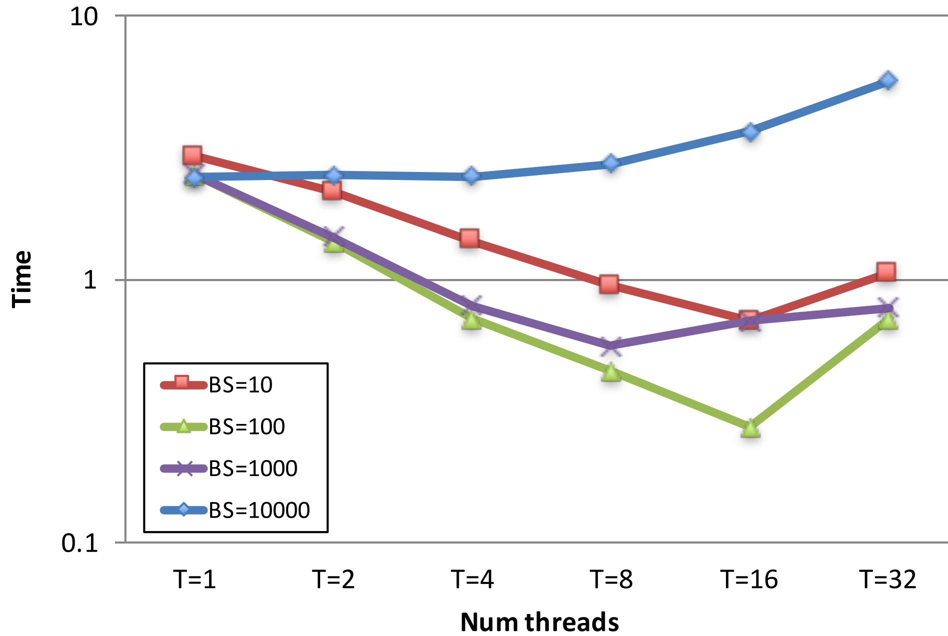
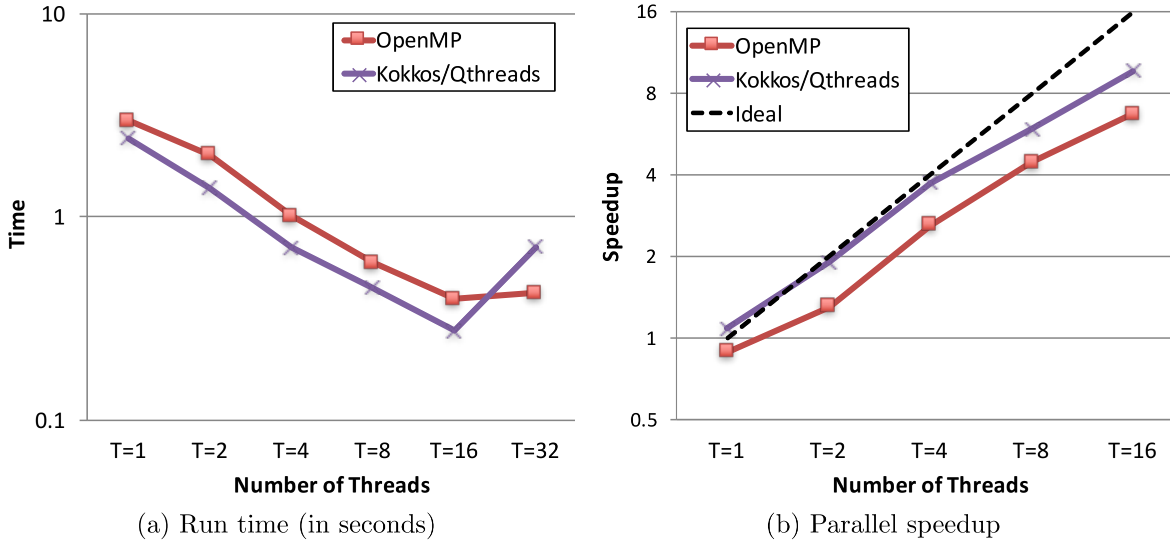


Figure 6.4: Run times (in s) miniTri run times (in s) for Oregon-1 graph for Kokkos/Qthreads implementation for different block size.

Fig. 6.5 shows a comparison of performance for the Kokkos/Qthreads task-parallel miniTri implementation with the baseline OpenMP data-parallel miniTri implementation as the number of threads are increased (where the best block size is chosen for each implementation). The Kokkos/Qthreads implementation significantly outperforms the OpenMP implementation, with a reduction in run time of up to 30% (Fig. 6.5a). Surprisingly, there is some improvement even for one thread. Fig. 6.5b shows the parallel speedup for the two implementations relative to our best serial linear algebra-based implementation. Neither implementation exhibits linear scaling on this challenging dataset, with the Kokkos/Qthreads scaling slightly better than the OpenMP implementation.

Fig. 6.6 shows a comparison of the Kokkos/Qthreads and the OpenMP implementation for the email-Enron graph. Although there is not a significant difference for one thread, the Kokko/Qthreads implementation requires less run time than the OpenMP implementation when more threads are used (Fig. 6.6a). The Kokkos/Qthreads implementation scales close to linearly up to eight threads, but drops off for 16 threads (Fig. 6.6b). The OpenMP



(a) Run time (in seconds)

(b) Parallel speedup

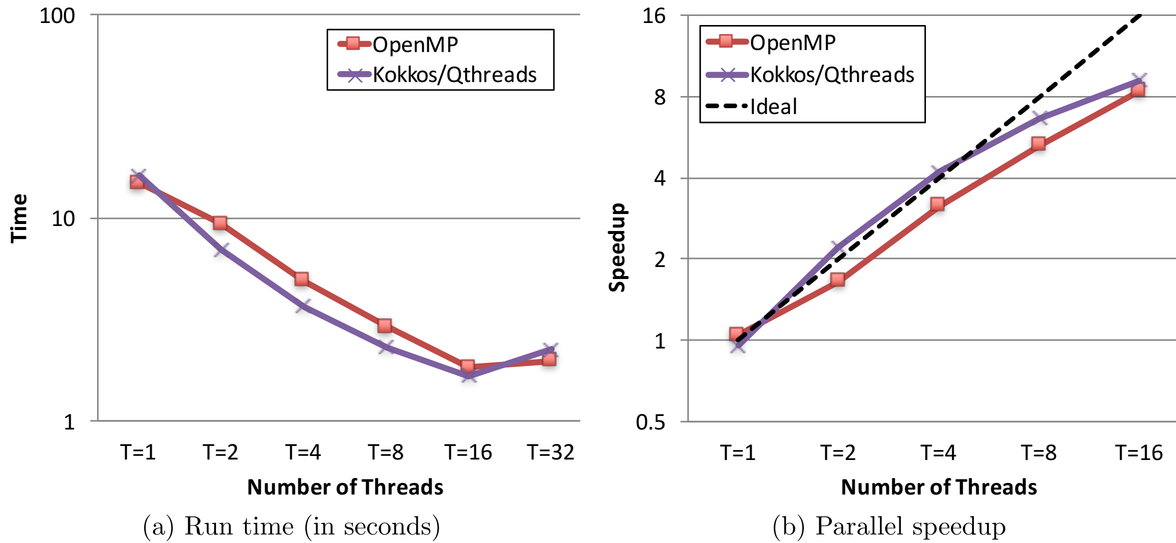
Figure 6.5: Comparison of OpenMP and Kokkos/Qthreads miniTri implementations for Oregon-1 graph (best block size for both methods shown).

implementation scales not as well for this problem across the number of threads, but is only slightly slower than the Kokkos/Qthreads implementations for 16 threads.

In Fig. 6.7, we see similar trends for the ca-AstroPh and the com-Youtube graphs when comparing the Kokkos/Qthreads implementation to the OpenMP implementation. As the number of threads are increased, the run time decreases for both implementations up to 16 threads. For both graphs, we see slightly better scalability for the OpenMP implementation than the Kokkos/Qthreads implementation (although the difference is slight for 8 and 16 threads). The raw performance for the Kokkos/Qthreads implementation is slightly better than the OpenMP implementation for up to 16 threads for both the ca-AstroPh and the com-Youtube graphs.

6.5 Conclusions/Lessons Learned

The Graph BLAS standard, which supports the expression of graph algorithms in terms of linear algebra primitives, shows great promise for enabling the development of high performance graph analytics. However, there are performance challenges with this approach when implemented in a traditional bulk synchronous manner, as demonstrated by the miniapp miniTri, including poor load-balancing. Another challenge demonstrated by this approach to miniTri is the explosion of temporary state (storing all triangles) that occurs as a result of the high-level linear algebra constructs. A task-parallel approach outlined in this section and in previous work [51] is a promising solution to these Graph BLAS challenges. It is also important to note that this miniTri task-parallel implementation has started to significantly impact the Graph BLAS standardization process. In recent Graph BLAS Forum discussions



(a) Run time (in seconds)

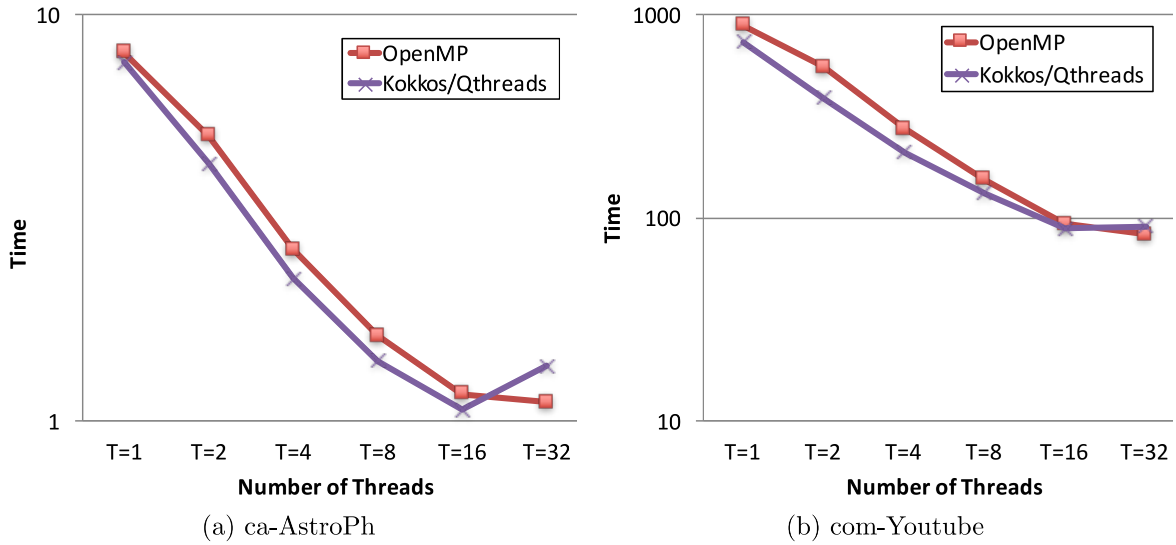
(b) Parallel speedup

Figure 6.6: Comparison of OpenMP and Kokkos/Qthreads miniTri implementations for email-Enron graph (best block size for both methods shown).

related to asynchrony, this miniTri implementation has served as an important, concrete example of why asynchrony is important and how to best support asynchrony in Graph BLAS.

In this section, we described how to efficiently implement a fully asynchronous, task-parallel approach to a linear algebra-based miniTri, where all unnecessary global barriers between linear algebra kernel calls have been removed. This task-parallel approach was implemented using the Kokkos programming model and its interface to the Qthreads task-parallel runtime system. We presented computational evidence that this Kokkos/Qthreads task-parallel implementation performs as well (and often better than) a more traditional data-parallel OpenMP implementation. We consider this a successful outcome since OpenMP has significant compiler based optimization that is not available to our Kokkos/Qthreads implementation, which relies on the asynchrony of this approach to make performance gains. With additional improvements to task-parallel runtime systems (reduction of overheads, etc.), we expect further performance gains in the future. In future work, we plan to move to a 2D distribution of our matrices, which should give us additional performance gains (by limiting the task dependences due to triangle edge degree calculations).

The asynchrony provided by a task-parallel approach also lays the groundwork to address the memory challenge to the linear algebra based approach, the explosion of intermediate results (triangles) during the triangle enumeration stage. To address this problem, a key insight is that task parallelism can be used to reduce the memory footprint of miniTri by prioritizing the k-count tasks. By prioritizing the k-count tasks, the k-count tasks will finish (once their dependencies have been met) before all the triangle enumeration tasks have completed. Once a k-count task has completed for its set of triangles, the memory for those triangles can be freed. In order to do this, we need the runtime system to support this



(a) ca-AstroPh (b) com-Youtube

Figure 6.7: Comparison of miniTri run times (in seconds) for ca-AstroPh and com-Youtube graphs: OpenMP versus Kokkos/Qthreads (best block size for both methods shown).

advanced resource management, including a sophisticated system for prioritizing these tasks. Although this runtime system support does not currently exist, there have been discussions with the Qthreads and HPX developers to add this support in the near future. In the big picture, a task-parallel approach with this resource-constrained scheduling would allow a Graph BLAS implementation of miniTri to analyze much larger problems, addressing a significant shortcoming in this linear algebra based approach. The goal is to use task-parallelism to enable additional performance and flexibility in the Graph BLAS approach and yield higher performing graph analytics.

Chapter 7

Conclusion

7.1 Goals and Key Results

A Sandia Computing and Information Sciences (CIS) strategic goal is "Sandia will lead the way in development of a unified Exascale technology base which supports effective architectures for both engineering and analytic applications." We have contributed to the realization of this goal by creating a dynamic task DAG capability that is portable across next generation platforms (NGPs). A unique features of this capability is hierarchical task-data parallelism where tasks may execute in parallel on disjoint thread teams and those thread teams are used to perform data parallel operation within a task. A breakthrough feature is this is the first time a dynamic task DAG capability has been implemented on a GPU architecture. We also proved suitability of the Kokkos programming model and implementation to the analytics domain by making significant progress in porting Sandia's multithreaded graph library (MTGL) and applying Kokkos to the triangle enumeration operation and mini-application. We also implemented a sparse matrix Cholesky factorization mini-application using the Kokkos task DAG capability, and this implementation is both portable and competative with the performance of architecture-specific and domain-specialized versions of the algorithm.

We created the hierarchical-parallel dynamic task DAG capability by integrating technologies from Sandia's Kokkos and Qthreads projects. Before this LDRD Kokkos was limited to data parallel patterns that were performance portable across NGP architectures. Similarly, Qthreads was limited to single thread tasks executing on multicore CPU architectures. Through this collaborative LDRD Qthreads' dynamic task DAG scheduling technology was applied to implement Kokkos' non-Qthreads back-ends, most noteworthy the GPU CUDA back-end. Likewise Kokkos' thread team technology was applied to prototype thread team tasks within Qthreads.

We initially planned to create a prototype porting of Sandia's multithreaded graph library (MTGL) to assess the suitability of Kokkos, with dynamic task DAG capability, to the analytics application domain. This prototyping effort was so successful in the early phases of the LDRD that the decision was made to proceed with fully porting MTGL to Kokkos. This porting effort is well underway and a follow-on project has been funded to complete the port of MTGL to Kokkos.

We initially planned to create a prototype sparse matrix Cholesky mini-application to evaluate the suitability and performance of the new dynamic task DAG capability compared to similar domain-specialized and architecture-specific implementations of this operation. The protyping effort has been so successful that application developers requested expedited access to the sparse matrix Cholesky factorization capability. This will continue through a follow-on project to develop and deploy sparse linear algebra kernels on NGPs.

7.2 Continued R&D

Performance evaluation and improvement

Capabilities created through this LDRD reflect the "final" results of several extensive research, prototype, and evaluation iterations. These capabilities are a proof-of-concept that hierarchical task-data parallelism is tractable and portable across diverse next generation platform (NGP) architectures. We are confident that these capabilities represent a solid foundation for development of application algorithms that require the task DAG parallel pattern. We also recognize that this R&D effort initiated under this LDRD is not "final" – that continued performance evaluation with broader application use cases and subsequent improvements are required. Several projects have committed to stewardship of subsequent R&D for Kokkos task DAG, Qthreads thread team task, MTGL porting to Kokkos, and sparse matrix factorization capabilities.

Final peer review – nomenclature revisions

Kokkos' task DAG API nomenclature will be revised as per recommendations from the final internal peer review (Section 2.3).

- `TaskPolicy` will be renamed to `TaskScheduler`.
- Arguments provided to the spawn and respawn functions that specify how a particular task is to be dispatched will be referred to as `task policy` parameters.

This revised nomenclature improves conformance to the vernacular for task DAG capabilities and alignment with the terminology use in Kokkos' data parallel dispatch functions.

We will analyze use cases with application users that motivate the request for user specified size of task thread teams. This research may result in enhancing Kokkos' task DAG capability as requested, may lead to revision of the use case algorithms, or could inspire an evolution of Kokkos' task DAG capability in an unforseen direction.

References

[1] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Proceedings of the 22nd International Symposium on Parallel and Distributed Processing*, ser. IPDPS/MTAAP, April 2008, pp. 1–8.

[2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. of Parallel and Distr. Comp.*, vol. 74, pp. 3202–3216, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2014.07.003

[3] Q. Meng, J. Luitjens, and M. Berzins, "Dynamic task scheduling for the Uintah framework," in *Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10)*, 2010. [Online]. Available: http://www.sci.utah.edu/publications/meng10/Meng_TaskSchedulingUintah2010.pdf

[4] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, "Programming petascale applications with Charm++ and AMPI," *Petascale Computing: Algorithms and Applications*, vol. 1, pp. 421–441, 2007.

[5] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "ScatterAlloc: Massively parallel dynamic memory allocation for the gpu," in *Proceedings of 2012 Innovative Parallel Computing*, 2012.

[6] "ScatterAlloc: Massively parallel dynamic memory allocation for the gpu." [Online]. Available: https://github.com/ComputationalRadiationPhysics/scatteralloc

[7] "halloc: A fast and highly scalable gpu dynamic memory allocator." [Online]. Available: https://github.com/canonizer/halloc

[8] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. mei W. Hwu, "Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines," in *CIT*, 2010.

[9] J. Feo, D. Harper, S. Kahan, and P. Konecny, "Eldorado," in *Proceedings of the 2Nd Conference on Computing Frontiers*, ser. CF '05. New York, NY, USA: ACM, 2005, pp. 28–34. [Online]. Available: http://doi.acm.org/10.1145/1062261.1062268

[10] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *Boost Graph Library: User Guide and Reference Manual, The.* Pearson Education, 2001.

[11] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[12] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–14.

[13] B. Barrett, J. Berry, R. Murphy, and K. Wheeler, "Implementing a portable multi-threaded graph library: The mtgl on qthreads," in *International Symposium on Parallel and Distributed Processing*, ser. IPDPS '09. IEEE, 2009.

[14] D. Bader, J. Berry, A. Amos-Binks, D. Chavarria-Miranda, C. Hastings, K. Madduri, and S. Poulos, "STINGER: Spatio-temporal interaction networks and graphs (sting) extensible representation," Georgia Institute of Technology, Tech. Rep., 2009.

[15] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[16] Y. Shiloach and U. Vishkin, "An o (logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[17] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[18] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[19] S. Maleki, A. Yang, and M. Burtscher, "Higher-order and tuple-based massively-parallel prefix sums," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2016, pp. 539–552.

[20] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

[21] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 219–228.

[22] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," in *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[23] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, p. 11.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[25] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, "A survey of direct methods for sparse linear systems," *Acta Numerica*, vol. 25, pp. 383–566, 2016.

[26] B. C. Gunter and R. A. van de Geijn, "Parallel Out-of-Core Computation and Updating the {QR} Factorization," *ACM Trans. Math. Soft.*, vol. 31, no. 1, pp. 60–78, mar 2005. [Online]. Available: http://doi.acm.org/10.1145/1055531.1055534

[27] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[28] E. Chan, F. G. van Zee, E. S. Quintana-Orti, G. Quintana-Orti, and R. A. van de Geijn, "Satisfying your dependencies with Supermatrix," in *2007 IEEE International Conference on Cluster Computing*. IEEE, 2007, pp. 91–99. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4629221

[29] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. van Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level Parallelism," *ACM Transactions on Mathematical Software*, vol. 36, no. 3, pp. 1–26, jul 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1527286.1527288

[30] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2014.07.003

[31] S. Rajamanickam, E. G. Boman, and M. a. Heroux, "ShyLU: A Hybrid-Hybrid Solver for Multicore Platforms," *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 631–643, may 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6267865

[32] A. George, "Nested Dissection of a Regular Finite Element Mesh," *SIAM J. Numer. Anal.*, vol. 10, no. 2, pp. 345–363, 1973.

[33] F. Pellegrini, "Scotch and libScotch 6.0 User's Guide," Universite Bordeaux I, Tech. Rep., 2012.

[34] D. Hysom and A. Pothen, "Level-based incomplete LU factorization: Graph model and algorithms," Lawrence Livermore National Labs, Tech. Rep. UCRL-JC-150789, 2002. [Online]. Available: http://www.cs.odu.edu/ pothen/Papers/ilu-levels.pdf

[35] E. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R. van de Geijn, "A note on parallel matrix inversion," *SIAM J. Sci. Comput.*, vol. 22, no. 5, pp. 1762–1771, 2001.

[36] R. A. van de Geijn and E. S. Quintana-Ortí, *The Science of Programming Matrix Computations*. www.lulu.com, 2008. [Online]. Available: http://www.cs.utexas.edu/ flame/-book/Docs/book.pdf

[37] J. I. Aliaga, R. M. Badía, M. Barreda, M. Bollhöfer, and E. S. Quintana-Ortí, “Leveraging Task-Parallelism with OmpSs in ILUPACK's Preconditioned CG Method,” in *Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 262–269. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2014.24

[38] T. A. Davis and Y. Hu, “The University of Florida Sparse Matrix Collection,” *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[39] K. Kim, S. Rajamanickam, H. C. Edwards, S. L. Olivier, and G. Stelle, “Task Parallel Incomplete Cholesky Factorization using 2D Partitioned-Block Layout,” arxiv.org/pdf/1601.05871, Sandia National Labs, Albuquerque, NM, Tech. Rep., 2016. [Online]. Available: https://arxiv.org/pdf/1601.05871

[40] J. D. Booth, K. Kim, and S. Rajamanickam, “A Comparison of High-Level Programming Choices for Incomplete Sparse Factorization Across Different Architectures,” *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 397–406, 2016. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7529895

[41] M. M. Wolf, H. C. Edwards, and S. Olivier, “Kokkos/qthreads task-parallel approach to linear algebra based graph analytics,” in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 2016, pp. 1–6.

[42] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, “Basic linear algebra subprograms for fortran usage,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[43] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[44] Tim Mattson et al., “Standards for graph algorithm primitives,” in *Proc. IEEE High Performance Extreme Comp. Conf.*, 2013.

[45] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011, vol. 22.

[46] A. Azad, A. Buluç, and J. R. Gilbert, “Parallel triangle counting and enumeration using matrix algebra,” in *Proceedings of the IPDPSW, Workshop on Graph Algorithm Building Blocks (GABB)*, 2015. [Online]. Available: http://gauss.cs.ucsb.edu/ aydin/triangles-gabb.pdf

[47] A. Buluç and J. R. Gilbert, “The Combinatorial BLAS: Design, implementation, and applications,” *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496 – 509, 2011. [Online]. Available: http://gauss.cs.ucsb.edu/ aydin/combblas-r2.pdf

[48] J. Kepner, W. Arcand, W. Bergeron, N. T. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Dynamic distributed dimensional data model (d4m) database and computation system." in *ICASSP*. IEEE, 2012, pp. 5349–5352.

[49] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for nosql databases," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 822–830.

[50] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[51] M. M. Wolf, J. W. Berry, and D. T. Stark, "A task-based linear algebra building blocks approach for scalable graph analytics," in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[52] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep*, 2009.

[53] "SSCA#2 v2.1 Specification," 2007. [Online]. Available: http://www.graphanalysis.org/benchmark/

[54] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, 2010.

[55] N. Wang, J. Zhang, K.-L. Tan, and A. K. Tung, "On triangulation-based dense neighborhood graph discovery," *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 58–68, 2010.

[56] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips, "Tolerating the community detection resolution limit with edge weighting," *Physical Review E*, vol. 83, no. 5, p. 056119, 2011.

[57] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11.

[58] "SNAP: the stanford network analysis project." [Online]. Available: http://snap.stanford.edu

[59] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories