Blaze-Tasks: A Framework for Computing Parallel Reductions over Tasks

PETER PIRKELBAUER, Department of Computer Science, University of Alabama at Birmingham, USA
AMALEE WILSON, Applied Computer Science Group, Los Alamos National Laboratory, USA
CHRISTINA PETERSON, Department of Computer Science, University of Central Florida, USA
DAMIAN DECHEV, Department of Computer Science, University of Central Florida, USA

Compared to threads, tasks are a more fine-grained alternative. The task parallel programming model offers benefits in terms of better performance portability and better load-balancing for problems that exhibit nonuniform workloads. A common scenario of task parallel programming is that a task is recursively decomposed into smaller sub-tasks. Depending on the problem domain, the number of created sub-tasks may be nonuniform, thereby creating potential for significant load imbalances in the system. Dynamic load-balancing mechanisms will distribute the tasks across available threads. The final result of a computation may be modeled as a reduction over the results of all sub-tasks.

This paper describes a simple, yet effective prototype framework, Blaze-Tasks, for task scheduling and task reductions on shared memory architectures. The framework has been designed with lock-free techniques and generic programming principles in mind. Blaze-Tasks is implemented entirely in C++17 and is thus portable. To load-balance the computation, Blaze-Tasks uses task stealing. To manage contention on a task pool, the number of lock-free attempts to steal a task depends on the distance between thief and pool owner and the estimated number of tasks in a victim's pool. This paper evaluates the Blaze framework on Intel and IBM dual-socket systems using nine benchmarks, and compares its performance with other task parallel frameworks. While Cilk outperforms Blaze on Intel on most benchmarks, the evaluation shows that Blaze is competitive with OpenMP and other library-based implementations. On IBM, the experiments show that Blaze outperforms other approaches on most benchmarks.

CCS Concepts: • **Computing methodologies → Shared memory algorithms**; **Concurrent algorithms**; Self-organization;

Additional Key Words and Phrases: Tasks, Reductions, Lock-free algorithms

ACM Reference Format:
Peter Pirkelbauer, Amalee Wilson, Christina Peterson, and Damian Dechev. 2018. Blaze-Tasks: A Framework for Computing Parallel Reductions over Tasks. *ACM Trans. Arch. Code Optim.* x, y, Article zz (December 2018), 25 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Many computational problems can be expressed in form of reductions over some input data. Ciesko et al. define a reduction operation as an iterative update to a result variable [Ciesko et al. 2015].

Authors' addresses: Peter Pirkelbauer, Department of Computer Science, University of Alabama at Birmingham, 1300 University Blvd. Birmingham, AL, 35294, USA, pirkelbauer2@uab.edu; Amalee Wilson, Applied Computer Science Group, Los Alamos National Laboratory, Los Alamos, NM, 87545, USA, amaleewilson@lanl.gov; Christina Peterson, Department of Computer Science, University of Central Florida, 4000 Central Florida Blvd, Orlando, FL, 32816, USA, clp8199@knights.ucf.edu; Damian Dechev, Department of Computer Science, University of Central Florida, 4000 Central Florida Blvd, Orlando, FL, 32816, USA, dechev@cs.ucf.edu.

res := op(res, expr)

`op` is an associative and commutative operator, `expr` is some expression that computes a partial result over some input data, and the result variable `res` must not occur in `expr`. Under such scenario, the computation can be parallelized by partitioning the data into tasks, solving each partition independently, and combining the results of all partitions.

Ciesko et al. list three common usage patterns for task parallelism and reductions [Ciesko et al. 2015]. (1) For-loop iterations (or the class of primitive recursive functions) use a a-priori bounded iteration space. Without control structures of greater generality, the computation can be split into n p uniform tasks, and the tasks distributed to p processors. (2) While-loop iteration (or the class of general recursive functions) use an iteration space of undetermined size, such as in graph traversals. (3) Recursive functions as used frequently in backtracking algorithms.

Although a bounded iteration space can also be hierarchically decomposed, this work focuses on problems that can be implemented using while-loop iteration and recursive functions. Under these classes, the iteration space is unknown and tasks may contain complex control structures that make task execution nonuniform. Thus the decomposition is challenging and may lead to load balancing problems that may become worse with coarser-grain problem decomposition. One approach towards solving such problems is to dynamically overdecompose the problem domain into tasks that can be solved independently by a number of available threads/cores. In order to load-balance, threads use work-stealing to obtain work from other threads. Alternatively, created tasks could be distributed to available threads using work-sharing [Thoman et al. 2018].

The notion of task parallelism has become prevalent since the wide-spread adoption of multicore systems. Many available frameworks support tasks. Although OpenMP initially focused on implementing parallel for-loops over a data set, tasks have been supported since version 3.0 [Ciesko et al. 2015]. The Intel Cilk languages [Frigo et al. 1998; Leiserson 2009] are extensions to programming languages in the C family which support the notion of task parallelism. Intel Threading Building Blocks (TBB) also supports the notion of tasks [Intel Corp. 2018].

When a task has completed its computation, the result can be combined into a global result in various ways. (1) The parent task can wait until a sub-task has finished its computation. Task parallel programming models typically offer such synchronized execution model, but in the context of parallel task reduction this model would lead to over-synchronization. (2) A single shared object can be used to accumulate the results of each task. Due to contention on a single shared object, this approach does not scale with increasing core count (though it would perform better than approach 1). (3) Each thread could combine all results of its solved tasks locally and reduce to a global result at the end. Framework support for parallel reductions over tasks varies. Cilk supports parallel task reductions in the form of reducer hyperobjects [Frigo et al. 2009]. OpenMP will introduce the notion of parallel task reductions with version 5.0 [Ciesko et al. 2015]. Users of earlier versions need to manually implement thread-local reduction. To our knowledge TBB's task mechanism (*i.e.*, `task_group`) lacks support for parallel reductions over tasks, though reducer objects could be provided by programmers. In addition to tasks, TBB offers other mechanisms for parallel reductions over data sets.

This paper introduces Blaze-Tasks, a simple generic framework for running parallel tasks and computing reduction operations. Users specify a task, a functor operating on tasks, and the number of threads being used. Blaze-Tasks are implemented portably in the C++ programming language [ISO/IEC 14882:2017(E) International Standard 2017] using threads as the underlying parallel abstraction. At the core, Blaze-Tasks use n lock-free queues, where each thread owns one queue. To support efficient work stealing, the queues are implemented as single-producer and multi-consumer queues. These queues use an asymmetric implementation that assigns different responsibilities to data structure owner

and task-thieves. By separating the responsibilities, the implementation can rely on weaker (atomic) operations that reduce the need for synchronization between CPU cores.

Blaze-Tasks are entirely implemented in modern C++ and unlike Cilk or OpenMP do not make use of language extensions and language specific compiler optimizations. Our evaluation is carried out on an Intel x86 dual-socket system with 20 cores in total and support for 40 hardware threads, and on an IBM Power 9 dual-socket system with 40 cores and support for 160 hardware threads. On Intel, Cilk exhibits the best performance on most benchmarks, followed by Blaze tasks and gcc's OpenMP. On IBM, Blaze-Tasks performs better on most benchmarks than the tested alternatives.

The contributions of this paper are:

(1) A generic lock-free design of task management and reduction
(2) The notion of asymmetric data structures to minimize synchronization overhead
(3) A portable and customizable prototype implementation in C++ and its evaluation on multiple architectures

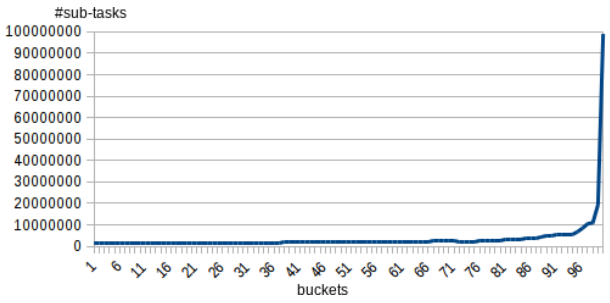
The remainder of this paper is outlined as follows. §2 provides the background of this work, §3 gives an overview of related work, §4 describes the design principles and implementation of our task management system in detail, §5 evaluates our approach by comparing it against other available systems, and §6 concludes and offers some ideas for future work.

2 BACKGROUND

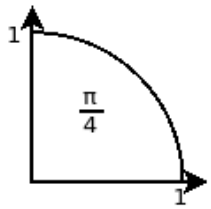
This section gives a brief overview of tasks, our prototype example, the Blaze library, and the C++ memory model.

2.1 Sample Problem - Computing π

Consider computing a numeric solution for a given integral. One approach is the trapezoidal method as outlined in Lambers lecture notes on computing adaptive quadrature [Lambers 2009, p2]. This algorithm begins by approximating a result for a given integral. The approximation is validated by computing another more fine grained approximation over two equally sized partitions. If the quality of the approximation is estimated to fall within a given tolerance threshold, a result is returned. Otherwise, two independent tasks for computing a numeric integral value for the two halves are created. The solution for the two halves can be independently sought and reduced to a global result.



(a) Distribution of subdivisions when approximating π through numeric integration using the trapezoidal method, with a tolerance threshold of 10^{-17}



(b) Approximate area under a quarter unit circle.

Fig. 1. Approximating π

```
double partialresult;
#pragma omp threadprivate(partialresult)

struct Task { double low, step, area; };
typedef double (Fun*) (double); // integratable function

double integrate(Fun fun, double lo, double hi, double tolerance) {
  double result = 0.0;
  double estimate = trapezoidal(fun, lo, hi−lo);

  #pragma omp parallel firstprivate(fun, lo, hi, tolerance, estimate) shared (result)
  {
    partialresult = 0.0;

    #pragma omp single
    #pragma omp taskgroup
    {
      integr_task(fun, tolerance, Task{lo, hi−lo, estimate});
    }

    #pragma omp atomic
    result += partialresult;
  }

  return result;
}
```

(a) Boilerplate code to start up adaptive numeric integration

```
double trapezoidal(Fun fun, double lb, double len) {
  return (len/2) * (fun(lb) + fun(lb+len));
}

void integr_task(Fun fun, double tolerance, Task task) {
  double half = task.step / 2;
  double a1 = trapezoidal(f, task.low, half);
  double a2 = trapezoidal(f, task.low+half, half);
  double area = a1+a2;
  double dif = abs(task.area − area);
  double tol = 3 * task.step * tolerance;

  if (dif < tol) {
    /* result is estimated to be sufficiently precise. */
    partialresult += area;
    return;
  }

  /* subdivide further. */
  #pragma omp task
  integr_task(f, tolerance, Task{task.low, half, a1}, s);

  #pragma omp task
  integr_task(f, tolerance, Task{task.low+half, half, a2}, s);
}
```

(b) Task to approximate an integral numerically

Fig. 2. Task parallel version of numeric integration in OpenMP

Depending on the underlying function, subdividing the integral can lead to nonuniform workloads. For example, approximating π by computing the area of a quarter circle (shown in Fig. 1b) leads to a much higher number of subdivisions on one side. A tolerance threshold of 10^{-17} yields the subdivisions shown by Fig. 1a. The figure subdivides the x-axis into 100 buckets and shows the number of segments that (partially) overlap with the bucket. The number of subdivisions increases from roughly 1.3 million per bucket on the left (bucket 1) to 99 million on the right (bucket 100), thereby causing significant imbalances in workload distribution.

2.2 Tasks

A task is a unit of computation that can be executed independently. Its execution may create more sub-tasks and yield some result. In many task-based systems [de Supinski, Bronis and Klemm, Michael (eds.) 2017; Leiserson 2009], a task may wait for the completion of its sub-tasks. In this work, we are interested in parallel reductions over all tasks, thus waiting on sub-tasks is not supported (at this time).

Consider the code in Fig. 2, which shows a task parallel implementation in OpenMP. Without the OpenMP pragmas, the code executes the computation sequentially. The approximation of the integral of function `fun` within the range between `lo` and `hi` is initiated by calling `integrate`. Integrate computes the area in the given range using the trapezoidal method and calls `integr_task` to validate the estimate. `integr_task` subdivides the range into two equal parts and approximates the same area. If the two approximations are within a tolerance limit, the result is added to the global variable `result`. Otherwise, the task is split into two sub-tasks for a more fine-grain computation.

In the OpenMP version, `integrate` creates a task group of threads. The main thread will fork new threads, which will work on available tasks, until they have been computed. The `taskgroup` construct synchronizes all threads by waiting until all tasks have been processed. In `integr_task`, the recursive subdivision creates two independent tasks (using `#pragma omp task`) that can be computed concurrently. In order to avoid contention from updating a single global variable, each OpenMP thread owns a variable, `partialresult`, where it aggregates the results of the tasks that it has

Table 1. Available Memory Management in the Blaze Library

Manager	Nonblocking	Description
Arena	no	releases memory only at quiescent periods
Epochs	no	uses thread vector clocks to track a thread's activity releases memory when threads are inactive or have progressed beyond the time when a node was removed from a data structure [Brown 2015; Pirkelbauer et al. 2017]
Publish and Scan	yes	tracks pointers in use by threads [Herlihy et al. 2005; Michael 2002]
Garbage Collection	yes	uses garbage collection to free unreferenced memory [Boehm 2002]

processed. At the end of the parallel region, the partial results of all threads are combined into a total value through the use of atomic operations.

Users can invoke numeric integration by supplying an integratable function as argument. For example, to approximate π, users model the circle formula to compute y for x.

```
double circle(double x) { return sqrt(1−x∗x); }
double pi = 4 ∗ integrate(circle, 0, 1, 0.001);
```

We argue that numeric integration is an example that leans itself towards a task-based model. It is a-priori unknown, where an integratable function requires subdivision. Thus, an approach that statically computes segments could suffer from significant load imbalances. A task parallel model remains flexible and immune against these imbalances (though fine-grain tasks may introduce other performance overhead).

2.3 The Blaze Library

The Blaze library [Pirkelbauer and Dzugan 2015; Pirkelbauer et al. 2017] offers scalable prototype implementations in C++ of concurrent utilities, including locks, containers, and memory managers. The Blaze library is built on generic principles [Dehnert and Stepanov 2000], where users can customize containers by plugging in different allocators and memory managers. The ability to customize Blaze's abstraction makes the library a suitable choice for environments that exhibit different safety and performance requirements. Table 1 shows available memory management schemes.
Memory management using epochs: Epochs keep track of threads starting and ending operations in shared data. In our implementation, a thread that accesses shared data increments its counter and increments it again when it finishes the operation. Other threads only read a thread's epoch counter and determine whether an epoch is active, and if a thread has made progress since a node was removed from the data structure.

When an object is removed from a data structure it is enqueued in a list of objects whose deallocation is pending. Objects in this list will be tagged with a timestamp indicating when the object was removed from the data structure. The timestamp is represented by a vector of epoch counters, $ts = (e_{t0}, \ldots, e_{tn})$. Initially, an object is untagged. When the number of untagged objects reaches a certain threshold τ, where $\tau >= |threads|$, a thread collects a new timestamp by reading all other threads' current epoch counters and associates the timestamp with yet untagged objects. The collected timestamp is also used to identify objects that can be freed. An object can be freed, if the comparison of its tag with the current timestamp shows that either every thread has advanced since the object was tagged, or a thread has been quiescent at the time when the object was tagged, $\forall_{0 \leq i < n},\ ts_i > tag(obj)_i \ \vee \ quiescent(tag(obj)_i)$.

Epoch management uses sequentially consistent operations to globally order the start of an operation and the start of object reclamation. The end of an operation uses release ordering to make sure that the effects of preceding operations become visible before an object can be freed.

Table 2. Memory Order Tags in C++11

Memory Order	Relationship among/between operations
seq_cst	Atomic operations are totally ordered. Non-atomic operations are partially ordered with respect to sequentially consistent atomic operations on the same thread.
release/acquire	Form a pair on the stored/loaded value. This guarantees that the reading thread sees all memory updates in the storing thread that occurred before the store tagged with release.
release/consume	Form a pair on the stored/loaded value. This guarantees that the reading thread sees all memory updates in the storing thread that occurred before the store and where there is a dependency relationship to the loaded value. In C++17 the use of consume is currently discouraged [Boehm 2016].
relaxed	No synchronization relationship.

2.4 The C++ memory model

With C++11, the C++ programming language introduced a relaxed memory model for concurrency. Memory locations subject to data races have to be of atomic type. A data race is defined as two or more concurrent memory accesses to the same memory location, where at least one of them is a write [Savage et al. 1997]. Atomic operations include load, store, and read-modify-write operations.

Programmers can exercise fine-grain control over memory ordering by tagging atomic operations (Table 2). The memory model offers four modes. Sequentially consistent operations (tagged `memory_order_seq_cst`) globally order an execution. Release/acquire (tagged `memory_order_release` and `memory_order_acquire`) and release/consume semantics (tagged `memory_order_release` and `memory_order_consume`) are pairs on stores and loads respectively. The tags guarantee that order dependencies (release/acquire) and data dependencies (release/consume) hold. Lastly, relaxed operations (tagged `memory_order_relaxed`) guarantee no ordering. The more relaxed an atomic operation the more aggressive optimizations (e.g., operation reordering) are available to the compiler and architecture.

The x86 architecture has a relatively strong memory model, where only sequentially consistent operations incur synchronization overhead in the form of a memory fence [Sevcik and Sewell 2011]. On the Power architecture, relaxed operations are cheaper than release/acquire operations, which in turn are cheaper than sequentially consistent operations [Batty et al. 2012]. An explanation of the C++11 memory model and more subtle details are described by the C++ standard [ISO/IEC 14882:2017(E) International Standard 2017], Boehm and Adve [Boehm and Adve 2008], and Williams [Williams 2012].

3 RELATED WORK

Cilk. Cilk is a parallel extension to the C programming languages [Blumofe and Leiserson 1999; Frigo et al. 1998; Leiserson 2009; Schardl et al. 2017] that consists of a compiler and a runtime system. Cilk pioneered many concepts available to task parallel programming. In Cilk 5 [Frigo et al. 1998] each hardware thread owns a deque to manage its tasks. The owner enqueues and dequeues from the near end, while the thieves dequeue from the far end. To avoid the overhead of using locks and expensive read-modify-write operations, the queue owner accessed its elements optimistically and checked afterwards that the access did not occur concurrently with a thief's access to the same element. Cilk 5's compiler creates two versions of each Cilk function, a fast version executed by the producer thread, and a slow version executed by thieves. The fast version elides synchronization due to task scheduling order guarantees. While implementation for multiple platforms exist, Cilk is not as widely available as OpenMP (e.g., not on IBM Power systems). While the development of Cilk continues with Open Cilk [Schardl et al. 2018], the support for Cilk has been deprecated by Intel and gcc.

An alternative deque implementation is presented by Arora et al. [Arora et al. 1998]. Their implementation is based on compare and swap to provide lock-freedom and pointer-tagging to prevent the ABA problem [Dechev et al. 2006]. Compared to our approach these deque implementations need to use sequentially consistent operations to synchronize accesses at both ends of a deque, which can be expensive on relaxed memory architectures. In addition, the described deque is implemented within a circular buffer, which - under some circumstances - puts some upper limit on the number of recursively spawnable tasks.

Intel Threading Building Blocks (TBB). TBB are a portable C++ library offering many generic abstractions suitable for parallel programming, including parallel algorithms and data structures, scalable memory allocators, and task schedulers [Intel Corp. 2018]. The use of tasks is supported through several methods. Tasks and task groups are the abstractions that are most similar to Blaze-Tasks in terms of its interface and design. In addition, TBB offers an algorithm for parallel reduction over data sets (*i.e.*, `parallel_reduce`).

The design of TBB's task abstraction and scheduling [Intel Corp. 2018] is very close to our design. Each thread owns its own task deque, where task-thieves remove tasks in a first-in first-out order, while the owner removes tasks in last-in first-out order (LIFO). TBB's focus with respect to tasks is on scheduling larger tasks. TBB uses C++ lambda functions [Järvi and Freeman 2010] as an abstraction for tasks. C++ lambda functions have the advantage that users can model heterogeneous tasks easier. Conversely, Blaze-Tasks store homogeneous task data that gets passed into a functor. This is not a limitation, however, as Blaze-tasks users could store C++ lambdas and define a functor to execute lambda functions.

Open Multi-Processing. (OpenMP) provides a common interface for programming shared memory multiprocessor systems [Ciesko et al. 2015; de Supinski, Bronis and Klemm, Michael (eds.) 2017]. Originally, OpenMP supported for-loop data-level parallelism, but its support for task parallel programs has been improving since version 3.0. OpenMP 5 will introduce support for task parallel reductions. OpenMP uses compiler directives to define extensions to host languages C/C++ and Fortran. OpenMP aware compilers translate the OpenMP directives into code for parallel execution. The combination of compiler plus runtime system enables compile-time optimizations, not available to library only implementations. OpenMP describes a common API that vendors can implement and optimize for specific target systems. Olivier et al. [Olivier et al. 2012] explore scheduling strategies on NUMA systems. Their work examines problems of task scheduling and work-stealing on NUMA architectures, which include cold cache misses and remote memory accesses. To improve scalability and reduce socket-to-socket communication, Olivier et al. suggest to use hierarchical task scheduling. For every shepherd there exists one LIFO queue that all threads use to share work. Thus work sharing occurs naturally per socket. Work stealing from other sockets is only attempted by the worker that finds the local task queue empty. This technique reduces inter-socket communication. Evans et al. discuss scheduling strategies for Chapel tasks on manycore systems [Evans et al. 2017]. They compare two existing strategies Sherwood and Nemesis and introduce a novel strategy Distrib. While Sherwood employs a hierarchical shepherd strategy for work stealing and LIFO order task processing, Nemesis, derived from the MPICH2 queuing system, uses a lock-free multi-producer single-consumer FIFO order queue. Nemesis minimizes contention at the expense of locality. Distrib is a novel strategy that improves Nemesis' approach with work stealing. In addition, Distrib uses condition variables to put threads to sleep when no work is available for stealing.

QThreads. is a portable open-source library that can be compiled on many architectures [Wheeler et al. 2008]. Qthreads offers a number of parallel abstractions, including support for waiting/non-waiting for sub-tasks and reducers (called sinc). QThreads offers users control over scheduler choice and shepherd/worker ratios.

Argobots. Argobots [Seo et al. 2018] is a portable library that offers several tasking capabilities in the form of user-level threads (ULT) and tasklets. While ULT maintain their own stack and context, tasklets are a more efficient and lightweight alternative that execute atomically on a thread's stack without context switching. Argobots gives users control to manage many aspects of runtime behavior, which makes Argobots a suitable library to work along-side other available parallel frameworks. Similar to Argobots tasklets, Blaze Tasks also do not have their own stack and execute atomically. In contrast to Qthreads, Argobots do not offer a sinc abstraction, which makes it challenging to use tasklets for reductions.

Other approaches. Muddukrishna et al. [Muddukrishna et al. 2016] use a locality aware runtime and user annotations to distribute data to different NUMA nodes. They introduce work-stealing and work-dealing algorithms that take queue sizes and node distance into account before stealing (dealing) tasks from (to) other nodes. Broquedis et al. explored tasks and data-flow information to efficiently steal tasks in the libKOMP library [Broquedis et al. 2012]. HotSLAW [jai Min et al. 2011] is a dynamic tasking library for UPC. HotSLAW works on distributed memory systems and makes task stealing decisions based on the memory hierarchy. HotSLAW considers vicinity to determine the victim and chunk size.

Compile-time optimizations. Static optimizations of task parallel programs have been explored by Thoman et al. [Thoman et al. 2017]. The authors discuss analyses to optimize the execution of parallel tasks. Parameters that are tuned statically include task spawning, tasks granularity, and an estimate of the stack size required per task.

4 DESIGN AND IMPLEMENTATION

This section describes our design principles, the interface to the Blaze-Tasks library, and its implementation.

4.1 Design Principles

We established the following principles to guide our design:

(1) *Low-overhead in the common case*: enqueuing to and dequeuing from the task pool should exhibit minimal overhead. This criteria is beneficial to the expected common case where threads have sufficient work to operate independently from each other (*i.e.*, without the need for task stealing). Little additional overhead cost should be paid when compared to a single threaded system. Since the reduction operation is associative and commutative, the tasks can be executed in any order.
(2) *Minimize the update of globally shared variables*: Frequent updates to single globally shared memory causes contention and consequently degrades performance [Herlihy and Shavit 2012]. Moreover, sequentially consistent atomic operations incur a significant overhead, because they require a total global ordering. A scalable design should be based on weaker atomic operations that entail less synchronization overhead.
(3) *Lock-free progress guarantee*: Lock-freedom requires that at least one thread out of many contending threads makes progress [Herlihy and Shavit 2012]. Frequently, the use of lock-free algorithms offers better performance, though many cases exist where attaining lock-free progress entails overhead in the common/single-threaded case. While we favor the use of lock-free algorithms, we do not argue for a completely nonblocking solution. Nonblocking solutions would require termination safety. Thread failure in the context of fork-join parallelism

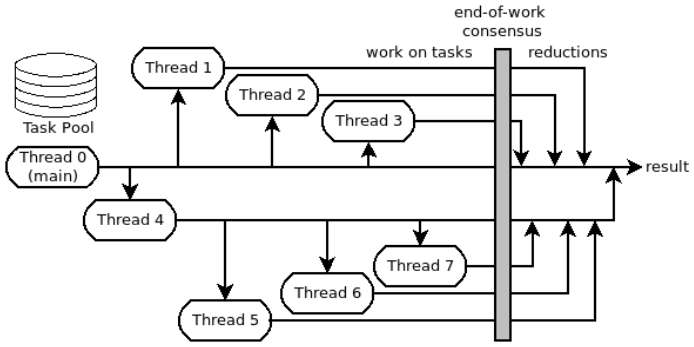


Fig. 3. Graphical Overview of Thread Creation and Reduction

would inherently lead to non-termination as the join operation would be compromised. Thus the use of blocking memory management in combination with lock-free algorithms is in order with our principles.

(4) *Cache locality:* Cache locality is critical to the performance of many algorithms. Any task scheduling system needs to provide mechanisms that ensure that data in cache can be reused by the same thread.

4.2 User Interface

Blaze-Tasks offers the function `execute_tasks` as the main interface to the task library.

```
template <class Functor, class Task>
auto execute_tasks(size_t numthreads, Functor f, Task t) → decltype(f(*new pool<Task>(t), t));

template <class Alloc, class Functor, class Task>
auto execute_tasks(size_t numthreads, Functor f, Task t) → decltype(f(*new pool<Task,Alloc>(t), t));
```

`execute_tasks` is a generic method that takes in three arguments. `numthreads` is the number of threads that will be created, `fun` is a generic functor that executes tasks. `fun` needs to support the function call operator() taking in a reference to a task pool (*i.e.*, `pool`) and a task. `fun` solves tasks of type `Task` and returns the result of the reduction. The result type is deduced from `fun` using the `decltype` specifier. Each thread initializes a result variable to the default element (e.g., 0 in case of a numeric type) and uses the `operator+=` to reduce the results first locally and at the end globally. User defined types can be used to initialize with a different neutral element or carry out a different reduction operation.

By default the memory allocated by the task pool is managed using epochs. Depending on the system, it may be more suitable to manage memory using an arena or a garbage collector. For these cases, a second version of `execute_tasks` allows users to pass in the memory manager as a template argument.

4.3 Implementation

This section describes our implementation in detail.

4.3.1 Task creation and reduction. The computation is started by invoking the function `execute_tasks(numthreads, f, task)`, which takes in the number of threads, a functor, and the initial task. `execute_tasks` runs on the main thread and is responsible for initializing the task pool with the initial task, for creating the worker threads, for executing part of the computation, and lastly for joining the threads and computing the reduction.

```
result_type reduction = result_type();
do {
  std::pair<T, bool> work = pool->deq();
  while (work.second) {
    reduction += fun(*tasks, work.first);
    work = pool->deq();
  }
} while (pool->has_work());
report_result_to_parent(reduction);
```

(a) Blaze's task loop

```
template <class Task, class Alloc>
struct pool {
  void enq(Task task); // enqueues a new task
  ...
};
```

(b) Task pool interface (user view)

Fig. 4. Task main loop and interface to create new tasks

Fig. 3 shows a graphical overview of the mechanism. The main thread initializes a shared task pool with the initial work and initiates the creation of the requested number of threads. If the number of threads is larger than a given threshold (*i.e.*, eight), the main thread tasks another thread with the creation of half of the remaining threads. Each created thread executes the task loop (Fig. 4a). The task loop takes a task from a pool and executes the task on its stack. The execution of a task may result in the creation of new sub-tasks, that will be stored in the pool. The execution of a task produces a result that is reduced locally (within a thread). A thread executes tasks (own or stolen) until no more tasks are available.

When the threads are in consensus that no more tasks are available, the result is reported back to the parent thread and the non-main threads terminate. The main thread returns the total result to the caller of `execute_tasks`. Fig. 4b describes the interface of the task pool relevant to users. The member function `enq` adds a new task to the shared pool.

4.3.2 Task Pool. In the previous section, we introduced the notion of a shared task pool. While several lock-free and wait-free data structures for the construction of multi-producer multi-consumer queues exist [Feldman and Dechev 2015; Michael and Scott 1996; Pirkelbauer et al. 2016; Yang and Mellor-Crummey 2016], we posit that many of these solutions incur a significant overhead (in the form of contention or the use of expensive atomic operations) even when a thread produces and consumes its own tasks. Thus, we opted for a design where the pool consists of n task queues where each of the n threads owns its own queue. When a thread requests a task from the pool, it first attempts to dequeue from its queue. If its own queue is empty, it tries to steal a task from other threads' queues.

In order to implement efficient local task queues, we introduce the notion of an asymmetric concurrent data structure.

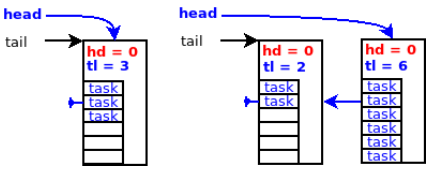
Definition 4.1. A shared concurrent data structure is called *asymmetric* if different threads take on different responsibilities with respect to the data structure.

Blaze-Tasks uses an asymmetric design that splits the responsibilities between queue owner and non-owners (task-thieves). The task queue uses a single-producer multi-consumer design. The owner is solely responsible for storing new elements in its queue and for managing its queue's memory. This distinction allows us to reap significant performance benefits for the owner's operations in terms of ensuring memory safety and the use of weak atomic operations.

Fig. 5b depicts the queue's design. To amortize dynamic memory management overhead, we use an unbounded queue of bounded queues. The outer unbounded queue is similar to Michael and Scott's queue [Michael and Scott 1996]. The design guarantees that at least one element remains in the queue, thus `head` and `tail` will never be `null`. The inner bounded queue uses a lock-free single-producer and multi-consumer scheme similar to a hybrid circular queue [Pirkelbauer et al. 2016], except that we do not reuse buffer elements.

```
template <class T, class Alloc>
struct taskQ {
  void enq(T el);
  std::pair<T, bool> deq(); // local dequeue
  std::pair<T, bool> deq_steal(); // dequeue from other thread's queue
  ...
}
```

(a) Operations of a task queue.



(b) Task queues owned by a thread. Other threads can access the queue to steal tasks. The black color indicates data that can only be accessed by the owner, blue color is data exclusively modified by the owner but read by other threads, while red indicates data that can be modified by all threads.

Fig. 5. Interface and design of a task queue

Fig. 5a shows the core interface of our task queue. `enq` and `deq` are the two operations that the queue owner uses to manipulate the content of its queue. `deq_steal` is used to dequeue a task from a non-local queue. The colors in Fig. 5b indicate which thread can access and modify the data (black = exclusive to owner, blue = data produced by the owner and read by other threads; red = data that can be modified by other threads).

The asymmetric distinction between queue owners and task-thieves distributes responsibilities between them. Owners are exclusively responsible for adding new elements to the queue, and for managing any memory allocation and reclamation. Task-thieves may dequeue elements (increment an inner queue's `hd`), but otherwise do not modify the queue's content. In turn, this offers opportunities for optimizing the queue owner's operations. (1) Since a task-thief only updates the inner queue's head (*i.e.*, `hd`), but does not modify the outer queue, the owner's enqueue and dequeue operations do not need to be memory managed. (2) The owner is the only producer, thus all intra-thread memory ordering relationships hold and the local dequeue can solely rely on relaxed memory operations.

Enqueue checks if the first inner queue has space available. If the queue is full a new outer queue node is added. Then the element is enqueued. Compared to a sequential queue, two modification were needed to ensure proper memory ordering. First, storing to the inner tail `tl` uses `release` to ensure that the preceding addition of the task is visible for all positions up to `tl`. Second, adding a new node to the outer queue needs to set the `next` pointer. To guarantee that the initialization of the new node is visible to other threads, the store is tagged with `release` ordering.

Dequeue removes the oldest element from the queue. To that end, it uses a lock-free dequeue implementation that optimistically reads a task pointed to by the current `hd` and attempts to compare and swap `hd` to one position afterward. If the compare and swap is successful, the task was dequeued. Otherwise, some other thread was successful, and the thread attempts to dequeue the task at the new `hd` position. If the inner queue becomes empty, dequeue attempts to remove from the next outer queue node. `deq` is also responsible for removing and freeing outer queue nodes whose tasks have been consumed and for updating the `head` pointer accordingly. All shared memory operations use `relaxed` memory ordering, except updates to `head` use `release` to ensure that at least the initialized node is visible to other threads.

When a node is removed from the outer queue, the owner is also responsible for freeing it. To this end, some concurrent memory management technique is applied to ensure that no other thread holds a reference to the memory. By default, we use epochs as a low-overhead mechanism [Hart et al. 2007], but users may plug in other available techniques, such as a garbage collector or an arena based memory technique as needed.

Since the data in a queue is produced by the owner, and the owner is solely responsible for managing queue memory, it can rely on intra-thread memory ordering and use only `relaxed` operations for dequeuing an element.

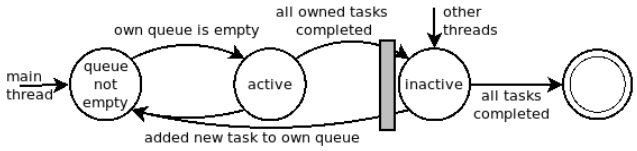


Fig. 6. State diagram of threads

Non-local dequeue steals tasks from other threads in a round robin fashion when the local queue is empty. A non-local dequeue reads a queue's `head` and attempts to dequeue an element from an inner queue. First, the dequeue reads the number of elements in the inner queue from `tl`. This is an acquire operation, that guarantees that all tasks stored up to `tl` are visible. A non-local dequeue optimistically reads the element at `hd` and attempts to compare and swap `hd` to one position afterward. If the compare and swap is successful, the task was dequeued. Otherwise, dequeue attempts to read the task at the new `hd` position. If the inner queue becomes empty, dequeue attempts to remove from the next inner queue. Non-local dequeues are not involved in memory management. Except for reading `tl` all operations use `relaxed` ordering. This is possible, because a task stored in an inner queue is never overwritten, and the memory management guarantees that any read from the location is served before the memory can be reclaimed and reused.

To manage contention on a single queue and to improve load balancing, we place a limitation on how often a thread can unsuccessfully execute the CAS loop on another thread's task queue, before it moves on to the next task queue. To do that, we use core proximity to determine how often a dequeue should be tried. We distinguish the following cases: two threads share the same core (in case of hyperthreading), two threads share the same socket (*i.e.*, they share some cache), and two threads execute on two different sockets. In order to better load-balance the tasks, we increase the number of attempts to steal from threads whose task pool is estimated to hold more than an average number of tasks, and we do not steal if the victim's pool contains significantly fewer than average number of tasks. In order to estimate the number of all tasks, a thread samples the available number of tasks and keeps a rolling average as it attempts to steal tasks. To this end, each thread keeps a rough count of the number of available tasks. The number is only updated when blocks of the outer queue are created or removed. Accesses to a task's counter uses relaxed memory operations, thus the count is considered an informed estimate.

On the IBM system, a thread `deq_steal` attempts the CAS loop six times (same core), four times (same socket), two times (other socket). If a task pool seems to contain a higher than average number of tasks, the number of attempts increases up to eight regardless of proximity before a thief gives up due to high contention. If a task contains less than a quarter of tasks in the rolling average, no task is stolen.

4.3.3 End-of-work consensus protocol. A set of threads that cooperate on solving a set of tasks have finished their work when no task is waiting in a task pool and when no thread is actively working on a task.

In a naïve scheme, we could simply count the number of created tasks and the number of processed tasks. When the two numbers are equal, all tasks have been processed and no more new tasks can be created. However keeping a global count of available tasks leads to contention on the counter. Thus, we use a different technique to keep track of available work. In our scheme a thread keeps track about its created and processed tasks.

The state diagram in Fig. 6 depicts the protocol. A thread is said to own a task, if a task was first enqueued in its task queue. For example, the first task is enqueued into the main thread's queue, thus the main thread owns the task. As long as a thread's *queue is not empty*, it dequeues from its own queue. When a task queue becomes empty, a thread

```
struct Task { double low, step, area; };
typedef double (Fun*) (double); // integratable function

double trapezoidal(Fun fun, double lb, double len) {
  return (len/2) * (fun(lb) + fun(lb+len));
}

struct compute_integral {
  Fun fun;
  double eps;

  compute_integral(Fun f, double tolerance)
  : fun(f), eps(tolerance)
  {}

  template <class P>
  double operator()(P& pool, Task task);
};

double integrate(Fun fun, double lo, double hi, double tolerance) {
  compute_integral comp(fun, tolerance);
  double estimate = trapezoidal(f, lo, hi−lo);

  return execute_tasks(20, comp, Task{ lo, hi−lo, estimate });
}
```

(a) Example of code using Blaze-Tasks

```
template <class P>
double
compute_integral::operator()(P& pool, Task task) {
  double dif, area, tol;

  do {
    double len = task.step / 2;
    double a1 = trapezoidal(fun, task.low, len);
    double a2 = trapezoidal(fun, task.low+len, len);

    tol = 3*task.step*eps;
    area = a1 + a2;
    dif = abs(task.area − area);

    if (dif >= tol) {
      pool.enq(Task{task.low, len, a1});
      task =  T{task.low+len, len, a2};
    }
  } while (dif >= tol);

  return area;
}
```

(b) Function application to approximate an integral numerically. The function creates tasks and adds them to the local taskpool. To improve cache locality, the last generated task is not added to the taskpool, but forms the continuation.

Fig. 7. Sample user code implementing numeric adaptive quadrature

remains *active*. A thread is *inactive* if all its owned tasks have been completed, either by itself or by some other thread that stole the task. Regardless of being in active or inactive state, threads attempt to steal tasks and work on them. When such a stolen task produces sub-tasks, a thread's queue becomes non-empty and the thread active. When all threads have become inactive, all created tasks have been processed.

At the beginning, only the main thread owns tasks. Some other thread will need to steal a task first, before it owns tasks. Each thread has a shared monotonic counter that counts the number of processed stolen tasks. Thus, when a thread finished a stolen task it updates the counter of the thread owning the task. To determine the number of inactive threads, Blaze uses a global counter. A thread modifies the global counter when it transitions from owning tasks to not owning tasks and vice versa. Only when all tasks have been processed, the count of active threads reaches zero. This technique guarantees that none of the worker threads terminate before all tasks are processed.

4.3.4 Example. Fig. 7 shows a Blaze-Tasks equivalent to the numeric adaptive quadrature algorithm implemented for OpenMP in Fig. 2. The algorithm is encapsulated by a functor, `compute_integral`, that stores the tolerance threshold and the integratable function. The numeric adaptive quadrature algorithm is implemented by `compute_integral`'s `operator()`. The main difference between the earlier OpenMP version and the Blaze version is that instead of splitting the approximation into two independent tasks, this version creates one task that is stored in the task pool and can be computed independently, and another task that serves as continuation. Once an integration task does not require further splitting, a result, `res`, is computed and returned. Then, the Blaze-Tasks main loop uses the result for a thread-local reduction and finds another task in the local queue or through work stealing.

The function `integrate` sets up the functor `compute_integral` for the Blaze Task framework, estimates the area for `fun` within the range from `lo` to `hi`, and starts the parallel reduction.

5 EVALUATION

This section evaluates Blaze-Tasks by comparing it to the state-of-the art frameworks OpenMP (as implemented by gcc 7.3, icc 15.0.3, and clang 6.0.1) and TBB (18 U3 and version distributed with icc 15.0.3), QThreads (checked out Aug. 2018, using Sherwood scheduler and 2 workers/shepherd, compiled with gcc 7.3), and Cilk (gcc 7.3 and icc 15.0.3). Qthread and Cilk were only tested on the Intel test system. For benchmark codes that were ported from the Barcelona OpenMP Test Suite (BOTS), we present the performance obtained for tasks without waiting (OpenMP) and the original BOTS implementation (BOTS).

We compare the runtime of our framework using an epoch-based memory management scheme. All test codes are written in similar style (*i.e.*, continuation loops are used by all codes or none, tasks do not wait on sub-task completion (with the exception of Cilk and BOTS implementations)). Since OpenMP does not support global task reductions currently (though OpenMP 5.0 will), we used manual reduction techniques [Ciesko et al. 2015] for OpenMP. The π and dot product benchmark use `threadprivate` variables, while the Fibonacci, N-Queens, and Unbalanced Tree Search benchmarks use a cacheline padded array that is indexed by the OpenMP thread number. For TBB, we provided our own reducer object, that builds on a cacheline padded array. The QThreads implementations follow our task-based programming style and use the provided *sinc* abstraction for reduction and termination detection. To manage the lifetime of heap allocated objects in task models that do not wait, we use C++ smart pointers. Since BOTS and Cilk versions wait until spawned tasks finish, the addition of memory management was not necessary with these programming models.

5.1 Experiments, Measurements, and Comparison

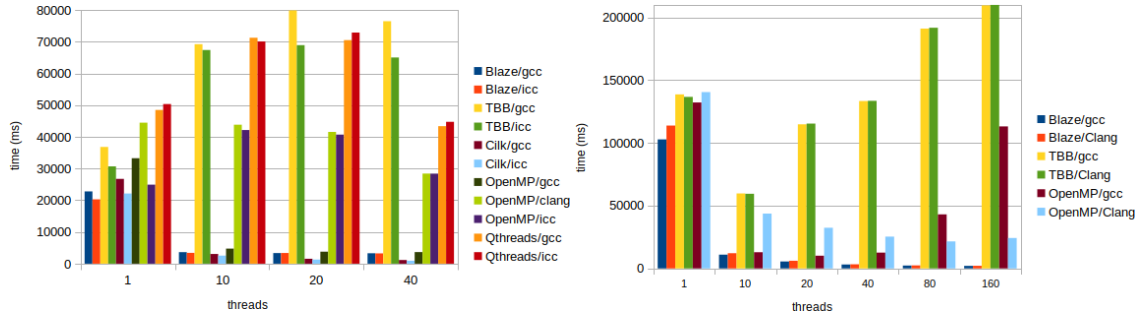
We tested the task systems using benchmarks that fall into three major categories: true parallel reductions, (*i.e.*, computation of π using a numeric adaptive integration algorithm, computing number of solutions to the N-Queens problem, computation of Fibonacci numbers, Traveling salesman (to some extent), computation of the Cosine between two vectors, Unbalanced Tree Search (UTS)); search problems (*i.e.*, Floorplan and Knapsack problem); and data parallel loops that were converted to tasks (*i.e.*, Alignment).

Table 3. The two main test systems

System	Sockets	Cores / socket	Threads / core	Clockspeed	OS
Intel E5-2660	2	10	2	2.6 Ghz	Red Hat 4.8.5-16
IBM Power 9	2	20	4	3.0 Ghz	Red Hat 4.14.0-49

We used two main test systems for evaluating the Blaze-Task library. Table 3 gives an overview of the system architectures. For each approach and test case, we varied the number of threads between one and the maximum number of threads supported in hardware. We ran each experiment 10 times, removed the best and worst result (as outliers) and averaged the remaining 8 measurements.

Approximation of π: The first test case is the approximation of π as described by our running example. We set the tolerance threshold to 10^{-17}, which produces 179,439,799 sub-segments. Fig. 8 shows the results. On Intel, Cilk (920ms, icc, 40 threads; 1150ms, gcc, 40 threads) runs about 3x faster than Blaze (3300ms, gcc and icc, 40 threads) and OpenMP/gcc (3600ms, 40 threads). The remaining approaches are significantly slower. On IBM, Blaze-Tasks (2050ms, gcc and clang, 160 threads) outperforms gcc 7.3's OpenMP by 5x (10100ms, 20 threads). Other approaches run significantly slower. To better understand the execution time, we used Likwid [Treibig et al. 2010] to gather secondary performance indicators for runs on the Intel system using 20 hardware threads. Table 4 reports the average runtime (measures TSC),



(a) Intel x86

(b) IBM Power 9

Fig. 8. Measured execution time for computing π with a numeric integral accuracy threshold of 10^{-17}. Numbers indicate measurements that exceed the time scale.

Table 4. Secondary performance measurements for computation of π on an Intel E5-2660 dual-socket running 40 threads.

	Runtime (s)	Unhalted runtime (s) (s)	Cycles w/o execution (%)	CPI
Cilk/gcc	1.18	1.26	22.1	1.3
Blaze/gcc	3.6	1.07	44.6	1.9
OpenMP/gcc	3.3	3.6	76.6	4.6
OpenMP/clang	27.4	28.0	89.0	7.0
TBB/gcc	30.9	34.4	97.0	34.2
QThreads/gcc	44.9	50.0	95.1	19.9

Table 5. Secondary performance measurements for computing fib(40) on an Intel E5-2660 dual-socket running 20 threads.

	Runtime (s)	Unhalted runtime (s) (s)	Cycles w/o execution (%)	CPI
Cilk/gcc	0.7	0.5	12.1	0.5
Blaze/gcc	3.8	3.5	93.6	13.5
OpenMP/gcc	4.3	4.5	92.6	6.8
OpenMP/clang	40.8	43.6	92.2	6.4
TBB/gcc	29.7	29.6	90.1	32.2
QThreads/gcc	68.2	74.5	96.2	22.4

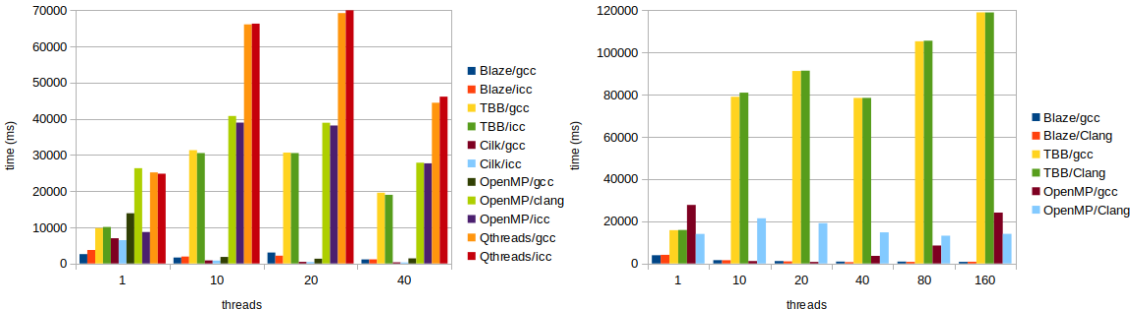
"unhalted runtime" (measures FIXC1), "cycles-no-execute" (computed by $\frac{\text{PMC3}}{\text{FIXC1}}$), and "cycles per instructions" (CPI) (computed by $\frac{\text{FIXC1}}{\text{FIXC0}}$ where FIXC0 is the number of retired instruction) to determine productive runtime and efficiency of an entire program execution.

The secondary indicators show that Cilk manages data more efficiently than alternative approaches, leading to a significantly larger percentage of unhalted runtime and fewer stalls.

Computing Fibonacci numbers: This benchmark expands the entire Fibonacci tree until a leaf node has either value zero or one. The sample code below shows Blaze-Task's implementation. Each task creates two sub-tasks, one of which is enqueued in the task pool, and the other one is executed as continuation.

```
template <class Pool>
size_t fib_task(Pool& pool, size_t num) {
  if (num <= 1) return num;
  while (--num > 1) pool.enq(num-1); // enqueues num-2; num-1 forms the continuation
  return num;
}
```

The result is computed by reducing the leaves. In our test, we computed the Fibonacci number for 40 which required a reduction over 165, 580, 141 leaves. Due to the computation's simplicity, this test exercises task creation and work stealing. The measured performance is shown in Fig. 9. On Intel, Cilk obtained the fasted performance (290ms, icc, 40 threads and 320ms, gcc, 40 threads) and ran 3.5x faster than Blaze (1140ms, gcc and icc, 40 threads) and 4.5x faster than gcc's OpenMP (1330ms, 20 threads). Other approaches are significantly slower. The secondary performance numbers in



(a) Intel x86 (b) IBM Power 9

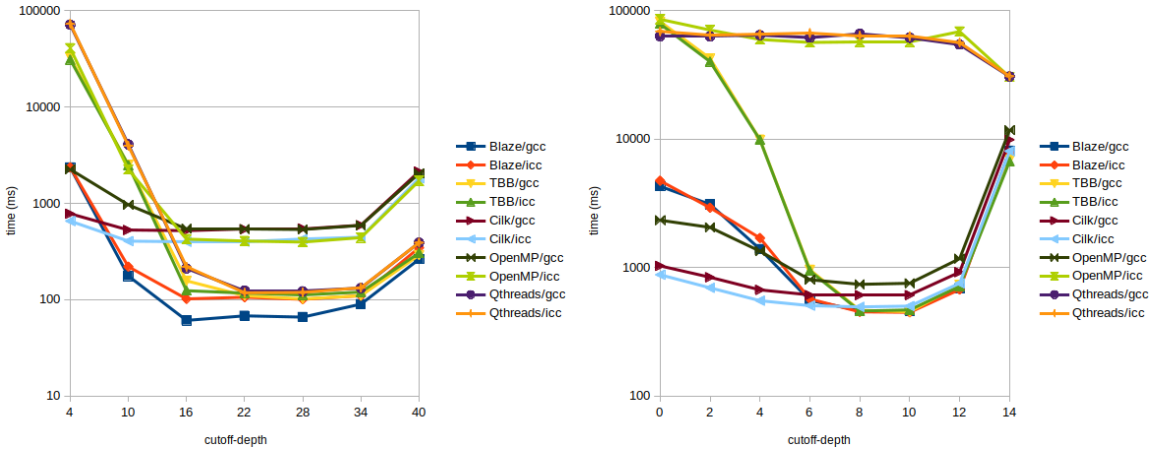
Fig. 9. Measured execution time for computing the Fibonacci number of 40.

Table 5 indicate that Cilk optimizes task scheduling and execution significantly better than other approaches. On IBM, Blaze had the shortest runtime (640ms, clang, 40 threads; 780ms, gcc, 160 threads), followed by gcc OpenMP (750ms, 20 threads). Other approaches were slower.

In order to compare the effect of different task granularity, we introduced a cutoff-level. Any input argument greater or equal to the cutoff value generates new tasks normally. Arguments less than the cutoff use local execution. The effect of the cutoff is larger leaf-tasks (several levels are computed within a single thread) and a reduced number of operations on the shared queue. Fig. 10a shows the obtained results. For low cutoff values, Cilk performs best. However, with increasing cutoff values, Blaze and TBB improve their execution time (70ms, Blaze/gcc, cutoff between 16 and 28; 105ms, Blaze/clang, cutoff between 16 and 28; 110ms, TBB gcc/icc, cutoff between 22 and 34). The overhead in Cilk is attributed to the slow-path in functions that could spawn, but actually do not spawn, and to the use of Cilk reducers for local computation. By eliminating the former, we could reduce the runtime to 350ms (icc, cutoff 16), and by eliminating both to 120ms (icc, cutoff 16). Note, the measured QThreads versions use the same improvements.

N-Queens: The next benchmark computes the number of valid solutions to the N-queens problem for a board size of 13. Results are shown in Fig. 11. On Intel, the best performance was obtained by Cilk (110ms, icc, 40 threads and 120ms, gcc, 40 threads), which was about 3x faster than gcc's OpenMP version (320ms, 40 threads) and 6x faster than the Blaze versions (650ms, gcc and icc, 40 threads). Other approaches were significantly slower. On IBM, the fastest runtime was obtained by Blaze (250ms, gcc and clang, 160 threads) followed by gcc's OpenMP (470ms, 20 threads). gcc's OpenMP version stops to scale beyond 20 threads. Other approaches were significantly slower.

We also experimented with cutoff values. Any task whose row number is less than the cutoff creates sub-tasks normally, while a task that exceeds the cutoff (closer to the leaves) elides task creation and uses a normal function call instead. Fig. 10b shows the obtained results for problem size 14 running on 20 threads. We varied cutoff levels from 0 (normal execution) to 14 (a single task). While Cilk obtains the best performance for low cutoff values, Blaze and TBB obtain the best overall runtime (about 450ms at cutoff level 10) as opposed to Cilk's best performance of about 490ms at a cutoff level of 8. Table 6 shows secondary performance indicators for cutoff of 10. Like the measured execution time, unhalted runtime and stall cycles of Cilk and Blaze match closely. Although, OpenMP/gcc has a better CPI and fewer stall cycles, it does not run as fast. Possibly, OpenMP/gcc uses more instructions related to task management. We were unable to gather performance data for TBB.



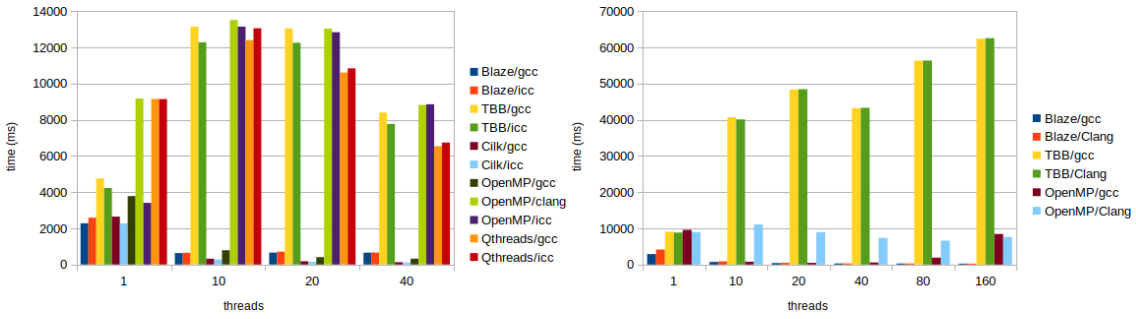
(a) Varies task granularity for computing fib(42). Results were obtained on the Intel system using 20 threads. Note, to emphasize the differences in runtime of faster approaches, the graph uses a logarithmic scale for the time axis.

(b) Varies task granularity for computing the N-Queens problem with 14 rows. Results were obtained on the Intel system using 20 threads. Note, to emphasize the differences in runtime of faster approaches, the graph uses a logarithmic scale for the time axis.

Fig. 10. Experiments with task granularity

Traveling Salesman: The next test case is a task-based approach to the traveling salesman problem. The input graph is represented as a matrix that indicates the weight of a path between two vertices. Each task entails traversals of the problem matrix (an $O(n^2)$ operation) and the creation of two sub-tasks. Thus, this is a computation bound problem. The benchmarks used problem size 12, which generates 20,582,108 sub-tasks. As shown in Fig. 12, the fastest performance on Intel was obtained by Cilk (icc, 846ms, 40 threads and 1255ms, gcc, 40 threads), followed by Blaze (1800ms, gcc, 40 threads and 1900ms, icc, 40 threads) and gcc's OpenMP (2380ms, 40 threads). On IBM, the fastest runtime was obtained by Blaze (1000ms, clang and gcc, 160 threads), followed by gcc's OpenMP (2400ms, 40 threads) and clang's OpenMP implementation (2900ms, 160 threads). On both systems, other approaches were significantly slower.

Floorplan: This benchmark was ported from BOTS, which in turn adopted it from the application kernel matrix by Cray. Floorplan uses a sequential outer loop that spawns tasks to explore possible solutions. The spawned tasks



(a) Intel x86

(b) IBM Power 9

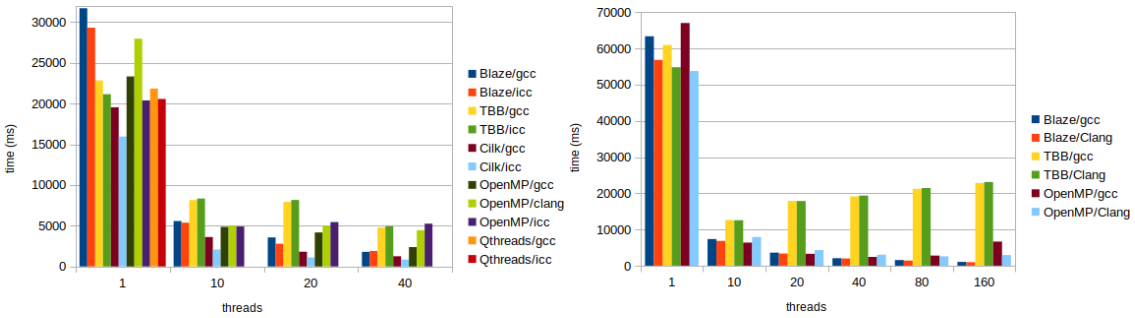
Fig. 11. Measured execution time for the N-Queens benchmark with problem size 13.

Table 6. Secondary performance measurements for N-Queens with cutoff of 10 on an Intel E5-2660 dual-socket running 20 threads. n/a indicates cases where data was not available.

	Runtime (s)	Unhalted runtime (s) (s)	Cycles w/o execution (%)	CPI
Cilk/icc	0.7	0.5	23.7	0.64
Blaze/icc	0.76	0.49	22.9	0.62
OpenMP/gcc	0.93	0.76	12.0	0.45
TBB/icc	n/a	n/a	n/a	n/a

Table 7. Secondary performance measurements for Alignment on an Intel E5-2660 dual-socket running 20 threads. n/a indicates cases where data was not available

	Runtime (s)	Unhalted runtime (s) (s)	Cycles w/o execution (%)	CPI
Cilk/icc	0.86	0.65	0.57	0.45
Blaze/icc	0.99	0.72	1.4	0.45
OpenMP/icc	n/a	n/a	n/a	n/a
OpenMP/gcc	1.10	0.66	0.86	0.41
TBB/*	n/a	n/a	n/a	n/a
BOTS/*	n/a	n/a	n/a	n/a



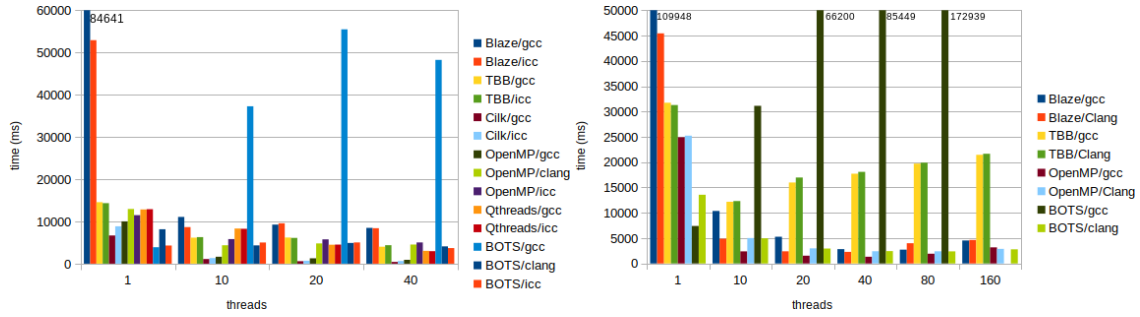
(a) Intel x86 (b) IBM Power 9

Fig. 12. Measured execution time for traveling salesman problem with problem size 12.

recursively invoke the outer loop with a smaller problem size. Results are shown in Fig. 13. On Intel, Cilk obtained the best performance (410ms, gcc, 40 threads and 620ms, icc, 40 threads), followed by gcc's OpenMP (900ms, 40 threads), and QThreads (2900ms, 40 threads). Blaze tasks performed relatively poorly, with 8300s running on 40 threads, although it needed slightly fewer tasks (19 million) than Cilk (22.7 million). Compared to the icc code, Blaze's gcc version performed noticeably slower when a small number of threads was used. By implementing copy and move constructors for the task class, gcc's runtime catches up to icc's. On IBM, the fastest execution was obtained by gcc's OpenMP version (1.3s; 40 threads) followed by Blaze/clang (2.2s; 40threads) and Clang's OpenMP and BOTS version (2.4ms; 80 threads).

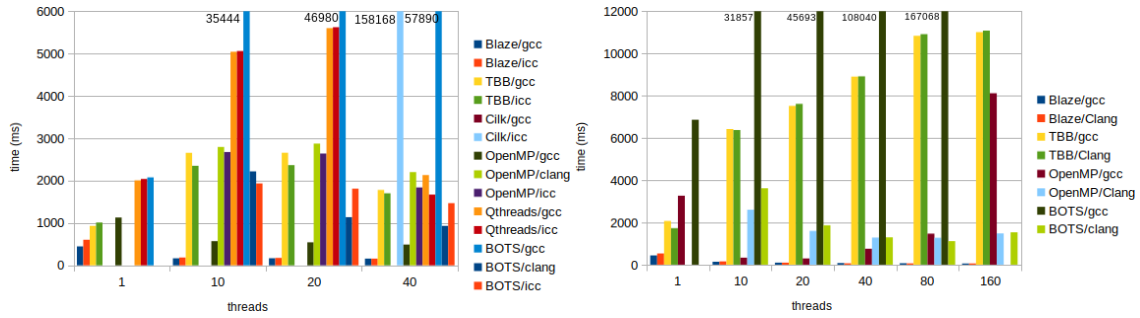
Knapsack. was ported from BOTS and originally developed for Cilk. Knapsack is task-based where each task waits until its sub-tasks complete. Each task computes an intermediate result. If this result is worse than the best solution found thus far, the task can be pruned. Otherwise, two sub-tasks are created. Results obtained for problem size 44 are shown in Fig. 14. On Intel, Blaze achieved the fasted performance (158ms; 40 threads), followed by gcc's OpenMP (.5s; 40 threads), and clang's BOTS version (.9s; 40 threads). Most codes finished in between 1.5s and 2.2s. On IBM, Blaze executed fastest (gcc and clang, 60ms, 160 threads), followed by gcc's OpenMP (300ms; 20 threads). The results obtained for this benchmark depend on how quickly a good solution can be identified. On Intel, Blaze needed about 11 million tasks (0.18s, using 40 threads; 61 million tasks/s), whereas Cilk needed about 209,160 million tasks (295s; 40 threads; 709 million tasks/s).

Alignment. was ported from BOTS, which adopted the code from the Application Kernel Matrix by Cray. The alignment benchmark consists of a data parallel loop. Additional tasks are spawned to handle complex computations



(a) Intel x86 (b) IBM Power 9

Fig. 13. Measured execution time for the Floorplan benchmark with problem size 15. Numbers indicate measurements that exceed the time scale.

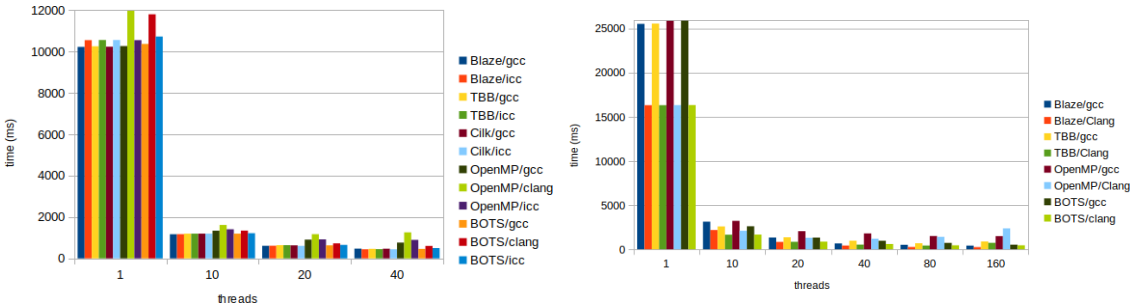


(a) Intel x86 (b) IBM Power 9

Fig. 14. Measured run-time for the Knapsack benchmark with problem size 44. Numbers indicate measurements that exceed the time scale; Experiments that did not find the solution within four minutes are omitted.

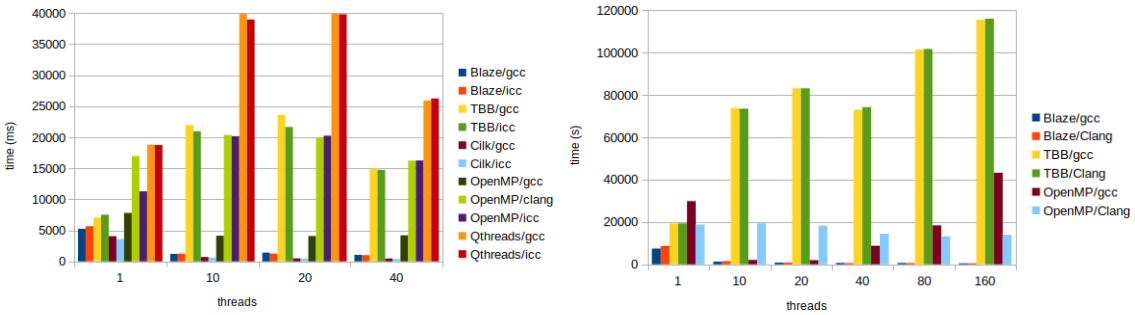
within the loop. In a task-only model, an application uses two different kinds of tasks. One task recursively subdivides the parallel iteration space (down to a single element). The second kind of task models the computation within the loop. The alignment test uses a significant amount of stack space, which caused QThreads to overflow although we experimented with larger QThread stack settings. The benchmark was executed with problem size 100 (the largest problem). Fig. 15 shows the measured runtime. The fastest time (about 450ms, 40 threads) was obtained by Blaze, Cilk, TBB, as well as gcc and icc's OpenMP version for BOTS. Slightly slower (1.5x-2x) were the remaining approaches using OpenMP (task only) and TBB. On IBM, Blaze obtained the fastest performance (240ms, clang, 160 threads and 410ms, gcc, 160 threads), followed by TBB (415ms, clang, 80 threads), followed by the BOTS versions (440ms, clang, 160 threads and 500ms, gcc, 160 threads). The OpenMP task-based versions were slower (1200ms, clang, 40 threads and 1480ms, gcc, 160 threads). In the test scenarios with fewer threads, the performance difference between clang and gcc is significant across all task frameworks. Using gprof, we traced back the slower runtime to a function `forward_pass` that allocates large arrays on the stack and uses a loop nest of depth two for non-affine accesses to a global array. It appears, that clang's O2 setting provides better optimizations for such a scenario than gcc's.

Table 7 shows secondary performance indicators measured on the Intel system. Since the problem size is relatively small, which benefits cache locality, the values obtained for Cilk, OpenMP tasks/gcc, and Blaze look similar.



(a) Intel x86 (b) IBM Power 9

Fig. 15. Measured run-time for the Alignment benchmark with problem size 100

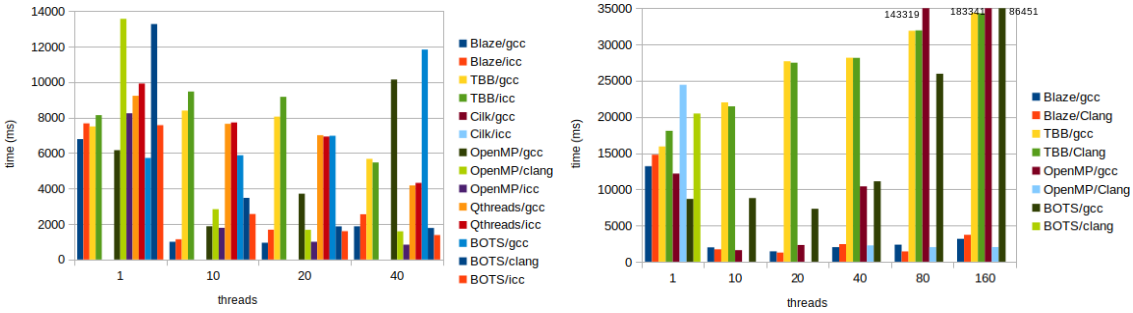


(a) Intel x86 (b) IBM Power 9

Fig. 16. Measured run-time for the cosine of two angles between two vectors, containing 100 million elements.

Dot product. computes the cosine of the angle between two vectors, $\frac{\vec{u} \bullet \vec{v}}{\|u\| \cdot \|v\}}$. The computation is implemented as reduction with three components, the element-wise product, and the element-wise squares of vectors u and v. The vectors in this benchmark consist of 100 million elements each. The computation space is recursively divided until the computation can be performed on a single element. On Intel, the fastest runtime was obtained by Cilk (390ms, icc, 40 threads and 460ms, gcc, 40 threads) followed by Blaze (1000ms, clang and gcc, 40 threads) and OpenMP (4000ms, gcc, 20 threads). On IBM, Blaze (500ms for gcc and clang using 160 threads) exhibited the fastest runtime followed by OpenMP/gcc (2000ms using 20 threads).

Unbalanced Tree Search (UTS). is a benchmark that evaluates the ability of task frameworks to dynamically load balance applications [Olivier et al. 2007]. UTS is an abstraction of search and optimization problems that require the exploration of a large state space. UTS' concrete implementation traverses all nodes in a search tree from the root to the leaves and counts the number of nodes. While the tree is dynamically constructed in a random fashion, determinism is achieved by controlling the number of subtrees generated by each node. Each node also performs work in the form of decoding its data and encoding its children's data. UTS leads to deeply nested spawns which may prevent some frameworks from completing the computation. We based our benchmark on the BOTS implementation, except for Qthreads, where we used the kernel from the Qthreads repository. Our performance benchmark was conducted using the tiny test case provided by the BOTS benchmark suite.



(a) Intel x86

(b) IBM Power 9

Fig. 17. Measured run-time for unbalanced tree search benchmark, using the tiny sample input. Numbers indicate measurements that exceed the time scale; experiments that did not find a solution within four minutes are omitted.

Table 8. Secondary performance measurements for UTS on an Intel E5-2660 dual-socket running 20 threads.

	Runtime (s)	Unhalted runtime (s) (s)	Cycles w/o execution (%)	CPI
Blaze/gcc	1.2	0.82	46.3	0.64
OpenMP/gcc	3.6	3.7	87.3	2.8
OpenMP/icc	1.5	0.94	34.9	0.64
BOTS/icc	1.6	0.8	53.6	1.8
TBB/icc	9.9	9.7	86.4	7.2
Qthreads/icc	6.8	7.2	90.7	4.9

Table 9. Secondary performance measurements for UTS on an Intel E5-2660 dual-socket running 40 threads.

	Runtime (s)	Unhalted runtime (s) (s)	Cycles w/o execution (%)	CPI
Blaze/gcc	1.96	1.2	64.8	1.5
OpenMP/gcc	10.8	12.0	94.7	11.2
OpenMP/icc	0.86	0.73	52.3	1.0
BOTS/icc	1.5	1.46	63.6	1.46
TBB/icc	5.3	5.9	94.1	9.6
Qthreads/icc	4.4	4.8	90.6	6.5

On Intel, the fastest performance was obtained by an OpenMP approach, where tasks do not wait for the completion of sub-tasks, (830ms, icc, 40 threads) followed by Blaze (950ms, gcc, 20 threads) and OpenMP as implemented by BOTS, where tasks wait for completion of sub-tasks, (1380ms, icc, 40 threads). The Cilk tests terminated with errors. On IBM, Blaze showed the best performance (1260ms, clang, 20 threads and 1430ms, gcc, 20 threads), followed by gcc's OpenMP implementation (1590ms, 10 threads), and clang's OpenMP version (2000ms, 80 threads). Several benchmarks did not complete within the four minute limit (e.g., BOTS/clang, OpenMP/clang for 10 and 20 threads).

Tables 8 and 9 show secondary performance indicators for experiments with 20 and 40 threads respectively. In the 20 thread case, Blaze/gcc has a larger percentage of unhalted runtime, while OpenMP/icc incurs relatively fewer stall cycles. BOTS/icc exhibits a fairly low ratio of unhalted runtime and relatively few stall cycles. In the 40 thread case, OpenMP/icc improves in terms of unhalted runtime although the stall cycles increase slightly, which in turn leads to the best measured runtime. In turn, Blaze/icc's unhalted runtime percentage and its stall cycles get worse, leading to a significant increase in runtime. BOTS/icc unhalted runtime improves while maintaining a similar stall cycle ratio, leading to the second best runtime.

Summary: We have tested several task-based frameworks on a variety of benchmarks using two dual-socket systems. On neither system there exists a single framework that performs better than all other approaches across all benchmarks. Nevertheless, Cilk performs best under most tested scenarios on the Intel system. The exceptions are cases where we introduced task generation cutoffs. While cutoffs seem to be a good tuning mechanism for problem domains that are regular and well understood, not all domains lean themselves towards the use of cutoffs. Examples include numeric

integration where the load imbalances depend on the integratable function, and unbalanced tree search. On the IBM system, Blaze tasks perform better than the tested alternatives on most benchmarks.

5.2 Usability

Our benchmarks were instances of the following parallel patterns:

- Parallel reductions: a task that cannot be solved directly will be divided into sub-tasks, which can be solved independently. The final result is computed through a reduction over the leaves in the task-tree. In our benchmarks, numeric integration (computation of π), Fibonacci numbers, N-Queens, dot product, unbalanced tree search, and Traveling Salesman are instances of this pattern.
- Search algorithms: many search algorithms sub divide the solution space recursively and could compute the final result as a min/max value reduction over the leaves in the task-tree. A common optimization applied by these algorithms is that tasks communicate the best solution found so far and prune the tree, as soon as the global result can no longer be improved. The global value also contains the final result. In our benchmarks, Knapsack, Floorplan, and Traveling Salesman (note, our TS implementation does not prune the search space) are instances of this pattern.
- Loop parallelism: data parallel loops are used to decompose a problem domain and solve each segment independently. Many frameworks (incl. OpenMP, TBB, QThreads) offer data-parallel abstractions. In a task-only system, the iteration space is recursively divided into sub-tasks, which compute independently. Of the tested benchmarks, alignment exhibits some properties of data parallel loops.

Task models that offer the ability to wait on spawned tasks are simpler to use in terms of resource management. Consider an object allocated by a task and provided as an argument to an arbitrary number of sub-tasks. If a thread can wait, the object can be deallocated by its owner. Conversely, if tasks cannot wait, programmers have to manage memory, using smart pointers (*i.e.*, `shared_ptr` in C++), garbage collection, etc.

The absence of waiting eliminates the need to synchronize tasks, thereby potentially speeding up the computation. This is evidenced by gcc's tasks implementation, where synchronization at the end of each task significantly slows down the execution, as measured for the Knapsack problem (BOTS vs OpenMP) and the computation of π (not shown). Even though Cilk elides synchronization in fast tasks (those executed by the producer), it still incurs synchronization overhead in slow tasks (executed by thieves).

While we have ported several benchmarks from BOTS, not all algorithms lean themselves to a simplified task model without waiting. For example, the FFT benchmark spawns tasks at two different levels. Each level requires waiting until the spawned tasks complete. Such implementation patterns are currently not supported by the Blaze library.

5.3 Discussion

Most schedulers utilize deque's for managing tasks. The deque data structure offers two main advantages. By using LIFO order for the owner, it is more likely that the data stored in the data structure is still in cache. In addition, the distinction between owner and thieves allows the owner to access the data without interference from task-thieves as long as a sufficient amount of tasks are stored in the data structure.

While lock-free deque data structures are known [Michael 2003; Sundell and Tsigas 2005], we opted for using a lock-free FIFO queue due to its simpler design that allows the use of relaxed memory ordering. In essence, we traded disjoint task removal in common cases and cache locality for a data structure that is almost as simple as a sequential implementation. To recuperate some cache-locality we ask users to design their algorithms using continuations (all

benchmarks other than Floorplan, Alignment, and UTS used continuations). A detailed analysis of the benefits and drawbacks will require experiments with these data structures within the same task framework.

With respect to our design goals, Blaze meets three of the four stated design criteria. The overhead for the common case where threads compute independently their own tasks is relatively low. However, many realistic task-based parallel programs will exhibit load-imbalances. Balancing the work-load with low overhead is another main factor that contributes to the overall runtime. Our implementation uses a lock-free design for a task pool to provide progress in the context of work-stealing.

Throughout our design, we have striven to minimize the update of global variables. In fact, the only globally shared variables updated by more than one thread is the counter associated with active threads, and the entry point for the epoch-based memory management (though this depends on the selected memory management). In order to attain cache-locality, our approach relies on programmers using continuation loops. In many cases, this kind of optimization can be automated by a using a semantically enhanced translator [Stroustrup and Dos Reis 2005].

Under most tested scenarios, Blaze-Tasks's performance is competitive when compared to other high-quality task management systems. In many test cases, Blaze-Tasks scale better when the number of threads exceeds the number of cores. We attribute this behavior to the lock-free nature of the implementation. Also, Blaze-Tasks seems to perform comparatively better on a relaxed memory architecture.

6 CONCLUSION AND FUTURE WORK

This paper introduced a lightweight library for task scheduling and computing parallel reductions over all tasks on shared memory systems. We introduced the notion of asymmetric data structures to reduce many sources of overhead in shared data structure design. The library is implemented in modern C++ and does not require any language extension or the use of task-specific compiler optimizations (though they may add additional benefits). The generic design allows for customizing the framework for specific environments, which is an important benefit to applications that use garbage collection or have other specific memory management constraints. The measured performance of the presented lock-free task management is competitive with other mature approaches.

The presented prototype constitutes a preliminary implementation that offers opportunities for further improvements. One such topic is the exploration of optimizations for NUMA architectures. Another topic is Blaze-Task library specific code optimizations that can be implemented through source-to-source translation. For example, task functions can be automatically transformed to work with continuations, relieving programmers from that burden.

A prototype of the Blaze library including Blaze-Tasks and the benchmark codes are available from our git repository at https://github.com/ppete/blaze.

ACKNOWLEDGEMENTS

This work has been supported by NSF grants CCF-1717515, CCF-1717635, CNS-0821497, and CNS-1229282. We thank the SimCenter at the University of Tennessee at Chattanooga for giving us access to their IBM Power 9 system. We are grateful to the anonymous reviewers for their constructive feedback and suggestions for improvement.

REFERENCES

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta.* 119–129.

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER. *SIGPLAN Not.* 47, 1 (Jan. 2012), 509–520. https://doi.org/10.1145/2103621.2103717

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. https://doi.org/10.1145/324133.324234

Hans-J. Boehm. 2002. Bounding Space Usage of Conservative Garbage Collectors. In *Proceeedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Hans-J. Boehm. 2016. *Temporarily discourage memory_order_consume.* Technical Report P0371R1. JTC1/SC22/WG21 C++ Standards Committee.

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

François Broquedis, Thierry Gautier, and Vincent Danjean. 2012. LIBKOMP, an Efficient openMP Runtime System for Both Fork-join and Data Flow Paradigms. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World (IWOMP'12)*. Springer-Verlag, Berlin, Heidelberg, 102–115. https://doi.org/10.1007/978-3-642-30961-8_8

Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*. ACM, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

J. Ciesko, S. Mateo, X. Teruel, X. Martorell, E. Ayguade, J. Labarta, A. Duran, B. De Supinski, S. Olivier, K. Li, and A. Eichenberger. 2015. Towards task-parallel reductions in OpenMP. In *International Workshop on OpenMP*. Springer, 189–201. https://doi.org/10.1007/978-3-319-24595-9_14

de Supinski, Bronis and Klemm, Michael (eds.). 2017. *OpenMP Technical Report 6:Version 5.0 Preview 2.* Technical Report. OpenMP Architecture Review Board.

Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2006. Lock-Free Dynamically Resizable Arrays.. In *OPODIS* (2006-11-29) *(Lecture Notes in Computer Science)*, Alexander A. Shvartsman (Ed.), Vol. 4305. Springer, 142–156.

James C. Dehnert and Alexander A. Stepanov. 2000. Fundamentals of Generic Programming. In *Selected Papers from the International Seminar on Generic Programming*. Springer-Verlag, London, UK, 1–11.

Noah Evans, Stephen L. Olivier, Richard Barrett, and George Stelle. 2017. Scheduling Chapel Tasks with Qthreads on Manycore: A Tale of Two Schedulers. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017 (ROSS '17)*. ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3095770.3095774

Steven Feldman and Damian Dechev. 2015. A Wait-free Multi-producer Multi-consumer Ring Buffer. *SIGAPP Appl. Comput. Rev.* 15, 3 (Oct. 2015), 59–71. https://doi.org/10.1145/2835260.2835264

Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and Other Cilk++ Hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 79–90. https://doi.org/10.1145/1583991.1584017

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA, 212–223. https://doi.org/10.1145/277650.277725

Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.* 67, 12 (Dec. 2007), 1270–1285. https://doi.org/10.1016/j.jpdc.2007.04.010

Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.* 23, 2 (2005), 146–196. https://doi.org/10.1145/1062247.1062249

Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming* (revised 1st edition ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Intel Corp. 2018. Reference for Intel Threading Building Blocks, Version 1.0. http://threadingbuildingblocks.org/. (2018). retrieved on May 5, 2018.

ISO/IEC 14882:2017(E) International Standard. 2017. *Programming Language C++.* JTC1/SC22/WG21 - The C++ Standards Committee.

Seung jai Min, Costin Iancu, and Katherine Yelick. 2011. Hierarchical Work Stealing on Manycore Clusters. In *In: Fifth Conference on Partitioned Global Address Space Programming Models. Galveston Island.*

Jaakko Järvi and John Freeman. 2010. C++ Lambda Expressions and Closures. *Sci. Comput. Program.* 75, 9 (Sept. 2010), 762–772. https://doi.org/10.1016/j.scico.2009.04.003

Jim Lambers. 2009. Lecture Notes on Adaptive Quadrature. http://www.math.usm.edu/lambers/mat460/fall09/lecture30. (2009). accessed on April 22, 2018.

Charles E. Leiserson. 2009. The Cilk++ Concurrency Platform. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09)*. ACM, New York, NY, USA, 522–527. https://doi.org/10.1145/1629911.1630048

Maged M. Michael. 2002. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*. ACM Press, New York, NY, USA, 21–30. https://doi.org/10.1145/571825.571829

Maged M. Michael. 2003. CAS-Based Lock-Free Algorithm for Shared Deques. In *Euro-Par '03, LNCS volume 2790*. 651–660.

Maged M Michael and Michael L Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 267–275.

Ananya Muddukrishna, Peter A. Jonsson, and Mats Brorsson. 2016. Locality-aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors. *Sci. Program.* 2015, Article 5 (Jan. 2016), 1 pages. https://doi.org/10.1155/2015/981759

Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2007. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing (LCPC'06)*. Springer-Verlag, Berlin, Heidelberg,

235–250. http://dl.acm.org/citation.cfm?id=1757112.1757137

Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. 2012. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *Int. J. High Perform. Comput. Appl.* 26, 2 (May 2012), 110–124. https://doi.org/10.1177/1094342011434065

Peter Pirkelbauer and Nicholas Dzugan. 2015. The BLAZE Concurrent Library. (2015). http://iprogress.cis.uab.edu/blaze.

Peter Pirkelbauer, Reed Milewicz, and Juan Felipe Gonzalez. 2016. A Portable Lock-free Bounded Queue. In *Proceedings of the 2016 16th International Conference on Algorithms and Architectures for Parallel Processing*.

Peter Pirkelbauer, Amalee Wilson, Hadia Ahmed, and Reed Milewicz. 2017. Memory Management for Concurrent Data Structures on Hardware Transactional Memory.. In *12th ACM SIGPLAN Workshop on Transactional Computing (Transact'17), workshop at the Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15, 4 (nov 1997), 391–411. https://doi.org/10.1145/265924.265927

Tao B. Schardl, I-Ting Angelina Lee, and Charles E. Leiserson. 2018. Brief Announcement: Open Cilk. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*. ACM, New York, NY, USA, 351–353. https://doi.org/10.1145/3210377.3210658

Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. *SIGPLAN Not.* 52, 8 (Jan. 2017), 249–265. https://doi.org/10.1145/3155284.3018758

S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castello, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel & Distributed Systems* 29, 3 (March 2018), 512–526. https://doi.org/10.1109/TPDS.2017.2766062

Jaroslav Sevcik and Peter Sewell. 2011. C/C++11 mappings to processors. (2011). www.cl.cam.ac.uk/ pes20/cpp/cpp0xmappings.html.

Bjarne Stroustrup and Gabriel Dos Reis. 2005. Supporting SELL for High-Performance Computing. In *18th International Workshop on Languages and Compilers for Parallel Computing*, Vol. 4339 of LNCS. Springer-Verlag, 458–465.

Hakan Sundell and Philippas Tsigas. 2005. Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap.. In *OPODIS 2004: Principles of Distributed Systems, 8th Int. Conf., LNCS, volume 3544*. 240–255.

Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. 2018. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing* 74, 4 (01 Apr 2018), 1422–1434. https://doi.org/10.1007/s11227-018-2238-4

Peter Thoman, Peter Zangerl, and Thomas Fahringer. 2017. Task-parallel Runtime System Optimization Using Static Compiler Analysis. In *Proceedings of the Computing Frontiers Conference (CF'17)*. ACM, New York, NY, USA, 201–210. https://doi.org/10.1145/3075564.3075574

J. Treibig, G. Hager, and G. Wellein. 2010. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. San Diego CA.

K. B. Wheeler, R. C. Murphy, and D. Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2008.4536359

Anthony Williams. 2012. *C++ concurrency in action: practical multithreading*. Manning Publ., Shelter Island, NY.

Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 16, 13 pages. https://doi.org/10.1145/2851141.2851168