Scalable and Locality-aware Resource Management with Task Assembly Objects

Miquel Pericàs
Chalmers University of Technology, Gothenburg, Sweden
Email: miquelp@chalmers.se

Abstract

Efficiently scheduling application concurrency to system level resources is one of the main challenges in parallel computing. Current approaches based on mapping single-threaded tasks to individual cores via worksharing or random work stealing suffer from bottlenecks such as idleness, work time inflation and/or scheduling overheads. This paper proposes an execution model called Task Assembly Objects (TAO) that targets scalability and communication-avoidance on future shared-memory architectures. The main idea behind TAO is to map coarse work units (i.e., task DAG partitions) to coarse hardware (i.e., system topology partitions) via a new construct called a *task assembly*: a nested parallel computation that aggregates fine-grained tasks and cores, and is managed by a private scheduler. By leveraging task assemblies via two-level global-private scheduling, TAO simplifies resource management and exploits multiple levels of locality. To test the TAO model, we present a software prototype called `go:TAO` and evaluate it with two benchmarks designed to stress load balancing and data locality. Our initial experiments give encouraging results for achieving scalability and communication-avoidance in future multi-core environments.

1. Introduction

To expose program concurrency several programming models have been developed including task-parallelism (e.g., Cilk), graph parallelism (e.g., TBB flow graphs), worksharing (e.g., OpenMP worksharing constructs) or mixed-mode parallelism. *Tasks* have become the unit of scheduling in such shared-memory models.

To ensure efficient resource management, runtimes need to intelligently schedule and map tasks to cores. In general, a runtime scheduler should target the following goals:

1. *Keep cores busy with work*, i.e. quickly transform application-level concurrency into hardware-level parallelism.
2. *Minimize runtime (library) overheads*, i.e. minimize the time performing runtime tasks (e.g., scheduling and mapping).
3. *Prioritize tasks on the critical path* (T_∞), mainly important when parallel slackness is low.[1]
4. *Minimize work time inflation* [16].
5. *Bound the memory* and *stack*, i.e., ensure that the working set does not overflow per core capacity, and limit the stack growth.

Developing an efficient runtime is however quite challenging. The difficulty stems from the fact that these goals are contradictory. For example, in order to keep cores busy parts of the computational DAG need to be moved to idle processors. This operation incurs runtime overhead, increases the working set and possibly disrupts locality, usually resulting in work time inflation. To reach exascale performance, computer architectures will need to undergo significant changes [3]. The forecasted increase in the number of cores per chip and deeper memory hierarchies will make the runtime's job even more difficult.

One important attribute that has an big influence on these goals is the task granularity, i.e. how much work is executed by a single core between two scheduling points. Improving parallel execution usually requires tuning the task granularity. The task granularity is usually tuned by *coarsening* the fine-grained tasks. Coarse tasks are then balanced dynamically via techniques based either on worksharing [1] or work-stealing [5].

Task coarsening simplifies task scheduling by reducing the activity of the runtime scheduler. However, the hardware mapping problem remains fixed at the single core granularity. Having large tasks execute on the small cores of future many-cores can easily overflow cache capacities and exhaust application-level concurrency. Furthermore, approaches based on work stealing do not ensure co-scheduling of tasks operating on shared data, leading to low cache efficiency. The solution that we propose in this paper is to not only coarsen tasks but to also *coarsen the hardware*, which we achieve by grouping worker threads.

The approach that we present is called *task assembly objects* (TAO). TAO aggregates tasks and worker threads to achieve scalable and locality-aware task scheduling on multi- and many-core hardware. From the scheduling perspective, a *task assembly* is an atomic parallel computation that allocates a set of worker threads for some amount of time. It can be seen as a two-dimensional task, spanning fine-grained tasks (dimension #1) and cores (dimension #2). Internally, an assembly object (i.e., the runtime instantiation of a task assembly) aggregates tasks, worker threads and a private scheduler. Locality is achieved by scheduling assembly objects to closely connected cores in the system topology. A TAO computation is managed by a two-level global (assembly-external) / private (assembly-internal) scheduler. Globally, a TAO computation is organized as a dependence graph of assembly objects. Task assemblies can efficiently encode parallel patterns such as `map` or `reduce`. We envision a flow-graph (DAG) of patterns as a programming style for productivity. Examples of this programming style are shown in Sections 5 and 6.

To validate this execution model, we have developed a TAO prototype runtime called `go:TAO` and implement two benchmarks: UTS and a parallel sort. The UTS analysis showcases TAO's ability to effectively schedule mixed-mode parallelism and adapt to load imbalance. The parallel sort explores hybrid (mixed) scheduling policies and multiple levels of locality. We compare the performance of `go:TAO` with several state of the art runtimes, includ-

[1] $T_P = \min_{\varphi} T(\varphi)$ denotes the execution time on P processors achieved with the shortest schedule φ.

ing TBB, MassiveThreads and Qthreads. Initial results suggest that `go:TAO` can efficiently schedule heterogeneous parallelism, while improving locality and reducing memory traffic.

This paper makes the following contributions:

- We propose Task Assembly Objects, an execution model which coarsens both workers and tasks to achieve scalable resource management (Section 3.1).
- We present the `go:TAO` prototype implementation, supporting the TAO execution model. `go:TAO` includes a dead-lock free method for scheduling task assemblies based on a streaming approach (Section 3.2).
- We port two applications to the TAO model and study their performance on the `go:TAO` prototype (Sections 5 and 6).

2. Scheduling, Mapping and Task Granularity

Task-parallel libraries usually manage the hardware by pinning a set of *worker threads* (e.g., pthreads) to the set of available cores. Scheduling and mapping, respectively, consists in choosing which task to execute and by which worker thread. Task granularity refers to the amount of work that a worker thread processes (non-preemptively) between two scheduling points. Coarse tasks reduce the scheduling frequency but constrain parallelism and may overflow the cache subsystem due to their larger working sets. Fine-grained tasks, on the other hand, expose large concurrency but increased scheduling overheads can deteriorate performance.

Task granularity is a program property, and cannot be set by an API except for very structured computations [1, 8]. Runtime library user guides therefore provide only very general guidance[2]. More detailed advice such as "*Create approximately 8 (M) times as many tasks as processors (P)*" [21] may come from the notion that, all tasks independent and equal, a parallelism of $M \times P$ results in a worst case load imbalance of $\frac{P-1}{(M+1)\times P}$, assuming a greedy scheduler [5]. Such advice tries to trade-off parallel slackness [22] and reduction of runtime overheads [20]. Following such guidelines results in a task granularity that depends only on P and is thus independent of the actual task characteristics. Consequently there is no guarantee that a program can efficiently use P processors.

3. Task Assembly Objects

3.1 The TAO Execution Model

TAO is designed to address the challenges of parallelism and deep memory hierarchies. The main idea behind TAO is to match hardware resources to application concurrency via a two-dimensional compute granularity, covering *work* (tasks) and *cores* (workers). The TAO execution model is shown diagrammatically in Figure 1. The 2D granularity is encoded into TAO's basic compute unit called *task assembly*, consisting of fine-grained tasks, a set of worker threads and a private scheduler. We refer to the instantiation of a task assembly as an *assembly object*. In TAO, coarsening refers to the grouping of several (fine-grained) tasks into a task subgraph that is scheduled as an atomic unit. TAO proposes to manage the resources via two levels of scheduling. Globally the system is governed by a dynamic scheduler. Dynamic scheduling is required to support conditional work generation and to adapt to load imbalances. Task assemblies feature an internal private scheduler that maps the work to *virtual cores*. From the point of view of the global scheduler, a task assembly is just a *wide* unit of computation requiring several cores. By mapping an assembly, the *virtual* cores are assigned *physical* core identifiers. Internally an assembly can be very

[2] for example, see https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Task-Based_Programming.htm

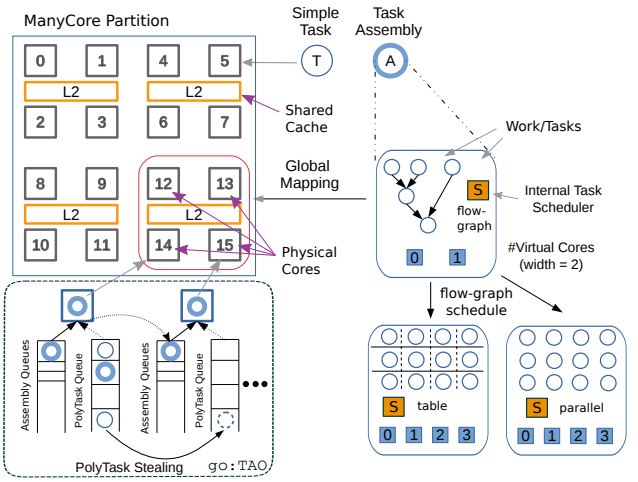


Figure 1. TAO Assemblies and the TAO Execution Model. Several assemblies are shown featuring different internal schedulers, such as `table`, `flow-graph` or `parallel`, and different resource widths. The box labeled `go:TAO` shows some details of the prototype implementation which is described in Section 3.2.

simple, like a single task or a table of tasks executed in lockstep, or it can feature a set of tasks with complex internal scheduling. In the extreme case, a TAO assembly can encapsulate a full OpenMP or TBB program, or even a TAO program itself. *Runtime nesting* provides separation of scheduling policies and places (i.e. topology partitions). TAO leverages nesting to address complex topologies and workflows. The separation of concerns allows TAO to support arbitrary forms of parallelism.

Assembly objects are low level concepts of the TAO model and are not expected to be programmed directly by the application developer. Assemblies are usually used to implement parallel design patterns, such as `map` or `reduce`. As a result a common programming style in TAO is to implement programs as flow graphs of parallel patterns. Yet assembly objects are much more general and can potentially be generated by the compiler via static analysis of task graphs or programmed directly by expert programmers (e.g., library developers). Assembly objects can potentially span from a single worker to all worker threads, and the work granularity can be as small as a single task or a complete application. In TAO, the ready subset of the assembly graph is initialized ahead of execution. At runtime, additional work can be dynamically generated to support conditional execution. The TAO execution model addresses the bottlenecks laid out in Section 1 as follows:

1. *Keep cores busy*: The two-level scheduling approach can generate parallelism quickly since every time an assembly is scheduled, several workers start executing in parallel. For example, 1024 cores can be allocated by scheduling 32 assemblies of width 32. Similarly, all the tasks inside an assembly are committed at bulk, thus reducing the pressure for the global scheduler to synchronize and reclaim resources.
2. *Minimize Runtime Overheads*: By grouping workers, TAO's global scheduler manages fewer discrete resources. In addition, by aggregating many tasks, an assembly executes autonomously for an extended period of time, which reduces the number of points at which global scheduling is invoked.
3. *Minimize Work Time Inflation*: Assemblies are scheduled onto worker threads that share levels of the cache hierarchy. This ensures that all tasks inside an assembly have access to nearby resources, which improves locality and reduces communication.

4. *Bound the Working Set* and *Stack*: Assemblies distribute a fixed working set across multiple cores. This reduces the total data to be loaded per chip and thus reduces pressure on the cache hierarchy. The explicit pattern flow graph style used by TAO is based on the notion of explicit continuations. Explicit continuations allow the stack to be recycled as soon as an object finalizes execution, which reduces the stack size.

TAO currently does not attempt to prioritize the critical path, although techniques such as adding priorities to objects in the flow graph would be simple extensions. Critical path scheduling becomes important when parallel slackness is not abundant. TAO currently focus on scenarios of high parallel slackness in which greedy schedulers have been proved to be efficient [4].

3.2 The `go:TAO` prototype

To examine the TAO execution model we are developing a proof-of-concept implementation named `go:TAO` with the goals of simplicity and portability but also reasonable efficiency. The TAO prototype is built on the `C++11` specification. Parallelism is managed via `C++11` threads and atomics. `go:TAO` consists of two parts: the global scheduler and the assembly execution. Pseudo-code for the worker loop of the global scheduler is shown in Listing 1. Assembly execution is handled by the private scheduler that is part of each assembly object and is hidden from the global scheduler. Encapsulating full runtimes such as TBB or OpenMP is currently not supported but such integration is planned as future work.

The global scheduler maps assemblies to worker threads. `go:TAO` supports assemblies of different *widths* (i.e. number of required cores, shown in Lines **26-34**) and also single-threaded tasks called *simple tasks* (Lines **8-12**). In the current prototype we use a set of distributed queues with *work-stealing*, a scheme that is known to scale [5] (Lines **37-40**). In `go:TAO` each worker thread manages two queues of ready objects. A main queue (called *PolyTask* queue) stores all ready objects, including both *assembly objects* and *simple tasks*[3]. We follow the Cilk5 stealing policy on top of this queue: objects are locally stored and retrieved using a LIFO policy but are globally stolen in FIFO order [10]. When a worker fetches a simple task object, the task is executed right away (Lines **8-12**). When a worker fetches an assembly object, it enqueues reference pointers into the *Assembly Queues* of each worker involved in its execution (Lines **13-23**). In the current implementation the worker selects a set of *assembly workers* sharing a level of the cache hierarchy to which the worker himself belongs (Lines **15**,**17**). This scheme exploits both spatial and temporal locality, since there is a high likelihood that the worker that makes the selection has recently executed one of the assembly's ancestors. Respecting the FIFO ordering is critical in the Assembly Queues to avoid deadlocks and provide a partial order of assembly objects. Figure 1 shows `go:TAO`'s PolyTask and Assembly Queues.

The *assembly queues* are checked with higher priority than the PolyTask queues. When a worker fetches an assembly from the *assembly queue*, it invokes its internal scheduler via the object's `execute()` method (Line **28**). From the view of `go:TAO`, what occurs inside an assembly is just a black box. Upon entering execution, control is given to the private scheduler which may decide to wait for other threads to join (synchronous execution) or proceed without waiting (asynchronous execution). `go:TAO` itself does by default not enforce any synchronization among assembly workers, which is handled exclusively by the assembly objects themselves. The assembly queue is managed in FIFO order. In the current development code, assemblies are inserted upon obtaining a lock to all participating assembly workers (Lines **15-19**). Locks and assemblies are acquired & inserted in FIFO order to avoid potential deadlocks. `go:TAO` can be configured to use either OS-based or busy-wait synchronization. We are mainly targeting HPC and embedded systems in which there is usually no competition from other applications. We thus default to using busy-wait synchronization. The PolyTask queue is an STL dequeue protected by per-thread spinlocks. The assembly queue uses a lock-free FIFO design which enables one parallel producer and consumer. `go:TAO` performs fairly well, however for wide assemblies with small granularities (e.g., $\lesssim 10\mu s$) the need to acquire one lock per worker introduces some idleness. We are currently looking for alternative mechanisms to ensure better scalability.

The programming interface of `go:TAO` is based on task assembly types (C++ classes) instantiated dynamically and inserted explicitly into the PolyTask queue of a particular worker, or in the parent's queue by default. This method provides an interface to partition the work across cores, similar to the *initial placements* method proposed by Acar et al. [2]. Dependencies among assemblies are described explicitly via a `make_edge()` method, a programming model similar to TBB Flow Graphs. Because continuations are explicit in this programming style, the stack can be recycled as soon as an assembly worker completes execution. When the last worker thread returns from the `execute()` method, the assembly object is also recycled. The assembly objects' interface consists of three methods: 1) the *constructor,* used to specify the *assembly width* and initialize internal data structures; 2) the `execute(thread_id)` method, invoked by each worker when the assembly is executed using the worker id (i.e., the physical core id) as argument; and 3) a `cleanup()` method, invoked when an assembly object becomes unreachable.

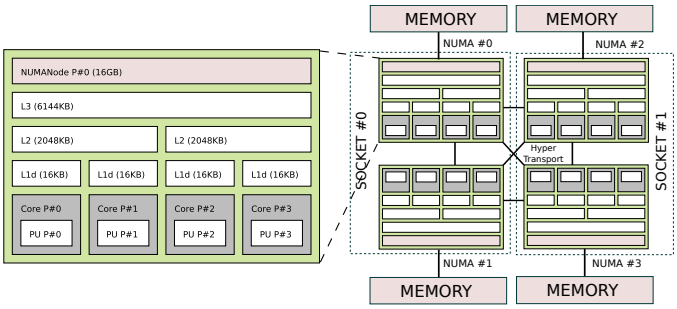


Figure 2. Topology of the Experimental Platform

4. Experimental Methodology

We evaluate `go:TAO` on an experimental platform consisting of two AMD Interlagos processors (Opteron 6220 @ 3.0 GHz) with two NUMA nodes per chip (total 4 NUMA nodes and 16 cores). Figure 2 shows the topology of a single NUMA node (as reported by `lstopo`) and of the full platform. We compare `go:TAO` with three state-of-the-art task-parallel schedulers: the Intel Threading Building Blocks v4.3 scheduler[4], the MassiveThreads library (July 2015 snapshot) [14] and Qthreads v1.10 [17]. All these runtimes are fine-grained, i.e. they schedule work at the individual thread and task. Lazy Task Creation schemes [13] are employed to amortize the cost of fine-grained task creation. Qthreads uses a hierarchical scheduler which combines work stealing and shared work queues. The characteristics of these runtimes are summarized in Table 1. `gcc v4.9` is used to compile all the tools and benchmarks.

[3] a `PolyTask` is a generic container for simple tasks and assembly objects

[4] https://software.intel.com/en-us/node/506295

Algorithm 1 Pseudo-code for `go:TAO`'s worker loop

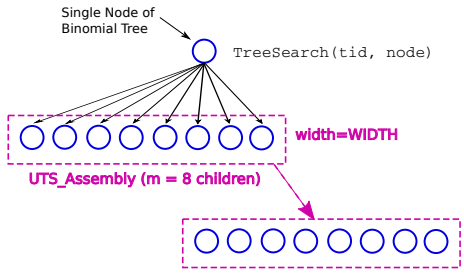
```
1  int worker_loop(int _nthr)
2  // worker pinned to the hardware thread == _nthr
3  {
4    PolyTask *task = nullptr;
5    while(1){
6
7      // 1. check forwarded path
8      if(task && task->is_simple()){
9        task->execute(task->args, _nthr);
10       task = task->commit_and_wakeup(_nthr);
11       continue;
12       }
13     else if(task && task->is_assembly()) {
14       // push work into assembly queues
15       for(i=first(_nthr,task->width); i < task->width; i++)
16             assembly_queue[i].lock();
17       for(i=first(_nthr,task->width); i < task->width; i++){
18             assembly_queue[i].push(task);
19             assembly_queue[i].unlock();
20         }
21       task = nullptr;
22       continue;
23     }
24
25     // 2. check if a ready assembly is in the queue
26     if(!task) task = check_assembly_queue(_nthr);
27     if(task){
28       task->execute(_nthr);
29       if(task->is_completed()){ // last assembly worker
30         ntask = task->commit_and_wakeup(_nthr);
31         task->cleanup(); task = ntask;
32         }
33       continue;
34      }
35
36     // 3. check for work local or global queue
37     if(!task) task = check_PolyTask_queue(_nthr);
38
39     // 4. try to steal from someone else's work queue
40     if(!task) task = steal_one_PolyTask();
41
42     // 5. check if the application has finished
43     if(pending_tasks() == 0)  return 0;
44     } // end while(1)
45 }
```

Runtime	Spawn Policy	Worker Queues	Stealing Policy
TBB	help-first	LIFO, distributed, 1 queue per core	FIFO, random core, single task
MassiveThreads	work-first	LIFO, distributed, 1 queue per core	FIFO, random core, single task
Qthreads	help-first	LIFO, shared, 1 queue per NUMA	FIFO, random node, bulk steal

Table 1. Characteristics of several runtime systems.

5. Mixed-mode parallelism

We begin by studying a graph search problem: the Unbalanced Tree Search (UTS) benchmark [15]. UTS is representative of a class of parallelism known as mixed-mode parallelism, which combines global task parallelism (e.g., OpenMP `#pragma omp task`) with task-level data parallelism (e.g., OpenMP `#pragma omp for`). TAO naturally supports *mixed-mode parallelism* with its two level approach. Task parallelism is supported by the global DAG while the data parallel pattern can be supported at the assembly level. Figure 3 shows how the UTS computation is organized into a pattern flow graph within `go:TAO`. The code snippet shows how assemblies are instantiated and placed into work queues.




```
void TreeSearch(int tid, struct args *in)
{
  // 1. compute hash
  rng_spawn(in->parent, in->node, in->spawnnumber);

  // 2. if non-leaf, create & place UTS assembly
  if(in->node->numChildren){
    UTS_Assembly *uts = new UTS_Assembly(in->node, WIDTH);
    gotao_place_work(uts, tid); // tid == thread id
  }
}
```

Figure 3. UTS assembly DAG (top) and source code corresponding to the computation of a single node (bottom).

UTS is a proxy for graph applications, which explore many nodes but 1) perform little computation on each node, and 2) have a small working set per node. We construct a UTS input consisting of a binomial tree with a small per-node "compute granularity" ($cg = 10$, about $3\mu s$ on the target system), and a probability of $q = 0.1249999$ that each node has $m = 8$ children. The root branching factor is $b_0 = 800$. This input results in a total of 148817 nodes to be visited. Using `DAG recorder`, a profiler included by MassiveThreads, we confirm that this input has enough concurrency to saturate the system. The task-parallel UTS implementations are based on the code provided with BOTS [9]. To target multiple runtimes we use a generalization of task-parallel interfaces developed at the University of Tokyo [12].

5.1 Scalability Analysis

Since UTS is dominated by very small computation in each node of the binomial tree, it becomes challenging to schedule and complete tasks efficiently. Dynamic schedulers, based either on a centralized queue or distributed queue with work-stealing, respectively suffer from queue contention and overhead in work stealing. Nodes in the UTS binomial tree have always $m = 8$ children nodes. We construct one assembly for each set of m children nodes. We initially consider only a fixed *assembly width* of four. This corresponds to the number of cores per NUMA node and is also the square root of the total number of cores in the system (16), which seems appropriate for two-level scheduling. To processes $m = 8$ nodes in a four-wide assembly object, we attach a simple table scheduler in which each virtual core processes two nodes. Since all nodes are independent, no synchronization is needed between the virtual cores, and worker threads are allowed to enter and exit the assembly asynchronously. Figure 4 shows the mean speed-ups achieved by `go:TAO` and the three fine-grained runtimes on several numbers of cores, along with error bars showing the standard deviation between runs. Unless stated otherwise, we perform 64 runs for all tests in this paper to reduce and study the effects of variability. In Figure 4 we introduce a small displacement in the x-axis so that error bars do not overlap.

Although the benchmark input has ample parallel slackness, the results show that extracting parallelism is challenging for the task-parallel runtimes. Qthreads in particular suffers from high contention on its shared *per-NUMA node* work queues, and also ob-

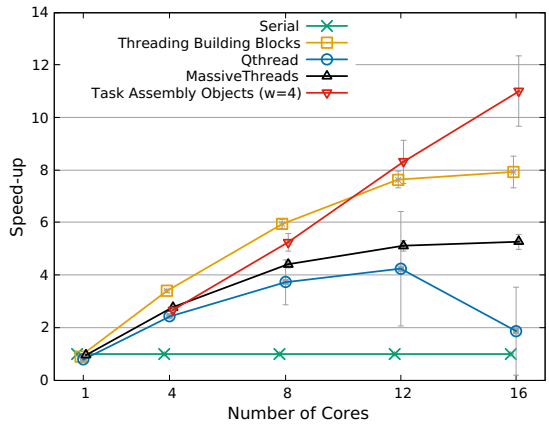


Figure 4. Scalability and standard deviation of UTS on the experimental platform for the task-parallel runtimes and `go:TAO`.

serves high variability at high core counts. A performance analysis via '`perf record`'[5] reported that at 16 threads, Qthreads spends most of the time in the runtime, trying to find work. MassiveThreads shows slightly better performance but saturates around $5\times$. Out of the fine-grained runtimes, TBB achieves the best performance, with a speed-up of $7.93\times$. Yet even for TBB the performance improvement over 12 cores is tiny (about $0.29\times$ compared to serial execution) indicating that performance is already saturated. The UTS tree is a self-similar tree that generates work randomly along its branches, which explains why the runtime is so often invoked to search for new work. The results show that the fine-grained schedulers' performance quickly saturates under these conditions. Overall, `go:TAO` achieves the highest speed-up at 16 cores ($11\times$), though interestingly it does not so at 8 cores and below. `go:TAO` is designed to scale to large numbers of cores. On small numbers of cores, the two-level scheduling approach is not yet effective, leading to higher idleness than fine-grained schedulers. As the core count increases, `go:TAO` however features a more stable trend, improving as much as $1.68\times$ from 12 to 16 cores compared to serial execution.

5.2 Assembly width

In the next experiment we analyze a range of assembly widths from one to eight. The schedule of the assemblies is shown in Figure 5. Aggregating N independent tasks into assemblies of size less than N increases per-thread work, which reduces global scheduling activity. However, it also limits the total parallelism. This performance trade-off is thus highly dependent on the implementation quality of the runtime the available concurrency in the application. Figure 6 shows the speed-ups over sequential execution achieved by the four assembly widths on 16 cores. The observation that *one-wide* and *two-wide* assemblies have better performance than *four-wide* and *eight-wide* assemblies points towards 1) inefficiencies in the runtime and 2) constructive cache sharing up to the shared L2 cache. With *one-wide* assemblies, a worker executes autonomously for about $24\mu s$, while for *eight-wide* assemblies the global scheduler is invoked every $3\mu s$. Scheduling assemblies currently requires sequentially acquiring one spinlock for each assembly worker. This introduces an overhead that affects the performance of wide assemblies. In terms of locality, the UTS assemblies can benefit from reuse in the children-parent accesses. Given that the working set is small, constructive cache sharing happens at the L2 cache level. This allows *two-wide* assemblies to outperform *one-wide* assemblies in this experiment. Wider assemblies, however, require data to be replicated into other L2 caches which degrades efficiency.

[5] https://perf.wiki.kernel.org/

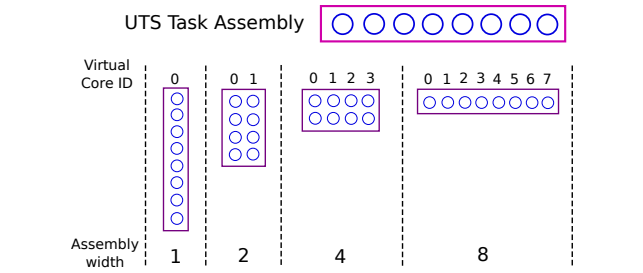


Figure 5. The implementation of the UTS assembly library.

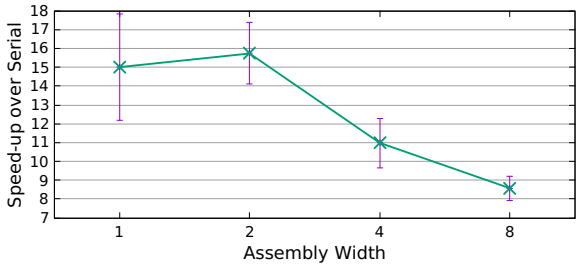


Figure 6. Average speed-up and standard deviation for several assembly widths.

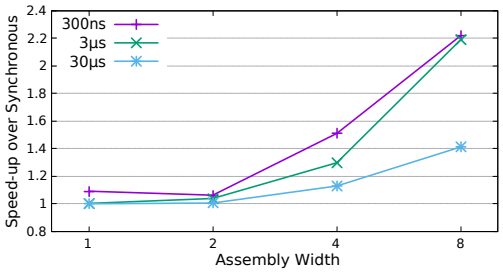


Figure 7. Speed-up of Asynchronous Assembly Execution vs Synchronous, for different node granularities and assembly widths.

5.3 Asynchronous Assembly Execution

Although TAO assemblies are conceptually 2D work units, implicit entry/exit barriers are not enforced by `go:TAO`. Since nodes in UTS are independent, we programmed the assembly workers to begin and finish execution asynchronously. To understand the impact of synchronous vs asynchronous execution, we evaluated both policies on uniform configurations of UTS for three node granularities ranging from $300ns$ to $30\mu s$. Figure 7 shows how the benefit of asynchronous execution grows with numbers of cores as barriers become more expensive[6]. For coarser granularities these overheads disappear, but the impact for wide assemblies is still visible and cannot be neglected.

6. Locality-aware Scheduling

We next analyze a parallel integer sorting benchmark derived from the BOTS `sort` benchmark, which is an implementation of the

[6] The improvement visible with *one-wide* assemblies is an artifact of using a barrier for synchronizing a single thread, which is obviously not necessary.

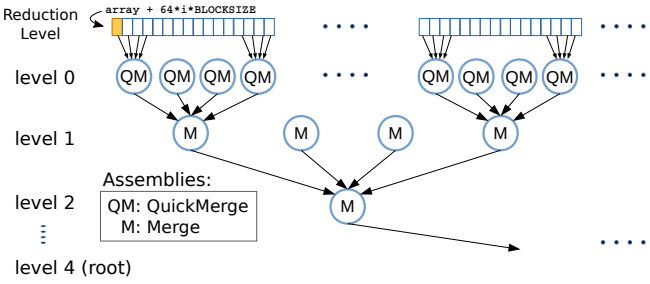


Figure 8. Parallel Sort assembly DAG

`Cilksort` algorithm. The `Cilksort` algorithm is composed of two stages: first an in-place sort (based on *quicksort* and *insertion sort*) is performed and next a double-buffered *mergesort* is conducted, in which segments of the array are merged in parallel. The original algorithm is recursive, and uses two cut-off parameters to limits the depth of the recursion. For `go:TAO` it is necessary to convert the recursion into a flow-graph with explicit continuations. We implemented a somewhat simplified version of the original algorithm in which each *mergesort* is split only once. This allows us to limit the size of the assembly library by reducing the number of patterns that need to be supported. We also implemented a task-parallel version of this simplified algorithm targeting the TBB and Qthreads runtimes. Except for the depth-based cutoff, this version is identical to the code provided in BOTS.

The `go:TAO` implementation of Cilksort uses two task-assembly types to implement the two phases of the program. An assembly *type* is a family of assemblies that realizes the same functionality but has different internal implementations. Figure 8 shows how the assembly objects are organized into a DAG. Figure 9 shows the implementations of the two assembly types: `QuickMerge`, which performs four in-place sorts and merges them, and `Merge`, which merges four sorted arrays. Two private schedulers, *table* and *dependence*, have been implemented. *Table* executes the internal tasks step by step, following a predefined table schedule and mapping. Each step is separated by a barrier involving all threads. *Dependence* is a dynamic scheduler in which all tasks are part of an explicit DAG which describes their dependences. The scheduler is implemented by using a single ready queue accessed by all worker threads. We evaluate assemblies of widths one to four. We stop at four since it is the maximum internal concurrency of both assembly types. We exploit the `go:TAO` placement API to distribute the assemblies across the NUMA nodes such that assemblies processing consecutive portions of the array stay on the same NUMA node. Given the reduction-like control flow of Cilksort, we conjecture that such distribution will minimize communication costs.

We run a sort of 32 million integers, requiring 256MB of storage for the array and 256MB for temporal buffering. At the first level we run 1024 in-place sorts, each of 32768 integers, distributed into 256 `QuickMerge` assemblies. The task granularity is relatively large, thus scheduling overheads are not problematic in this scenario. We run the experiments on all 16 cores to stress the runtime and experimental platform.

6.1 Uniform Assemblies

We first try to find the best configuration of assemblies when using the same private scheduler and assembly width for all levels of the reduction. Table 2 shows the speed-ups achieved by several uniform assembly implementations of the parallel sort with two private schedulers (*dep,* for dependence; and *tab*, for table) and assembly widths of *one, two* and *four*. The table shows that the best uniform assembly configuration is *dep-2*, i.e. two-wide assemblies with

	`dep-1`	`dep-2`	`dep-4`	`tab-2`	`tab-4`
`Speed-up`	5.39×	7.51×	7.34×	7.31×	7.42×

Table 2. Speed-ups compared to serial execution for uniform TAO configurations. `tab-1` is not shown since it is equivalent to `dep-1`.

dependence-based scheduling. Compared to four-wide assemblies, a width of two performs better at the higher levels of the task graph, since the flow graph concurrency is ample but the assemblies' internal concurrency is limited. The latter allows to keep cores busy over 90% of the time, as opposed to four-wide assemblies, where cores are idle for about 40% of the processor cycles. While *dep-2* can keep cores busy, in the two last levels it suffers since only five assemblies are ready to execute and thus only $4 \times 2 = 8$ (level 1) or $1 \times 2 = 2$ (root level) cores get allocated. Since these five assembly objects have a huge granularity, we expect to see improvements when mixing *two-wide* and *four-wide* assemblies.

6.2 Hybrid Assemblies

We now try out a hybrid scheme, in which the first levels of the reduction use *dep-2* assemblies, while the lower levels of the reduction use assemblies of width four. We evaluate two parameters: 1) the level at which we switch from *two-wide* to *four-wide*, and 2) table vs dependence scheduling of the *four-wide* assembly objects. The results in Table 3 show that the best performance is obtained when assemblies of width four are used only in the last level (root) of the reduction (configuration 4+1). Having two-wide or four-wide assemblies in the penultimate level has a small effect on performance even though it impacts the parallelism of the execution. At this level the working set of each assembly is 128MB (64MB sort array + 64MB temporal buffer). Such working set overflows the last level cache capacity by a considerable margin. As a consequence, performance is dominated by memory bandwidth and not by parallelism. However, at the root level, the performance of two-wide vs four-wide assemblies is quite different. We hypothesize that two cores are not enough to saturate the memory bandwidth of the experimental platform. As a consequence four-wide assemblies provide better performance.

Table 3 also shows that there is no big difference between dependence and table scheduling. In the last levels of the reduction the internal tasks of the assemblies are very big and the load is balanced, thus both *static* or *dynamic* schedulers perform similarly. Overall, hybrid scheduling allows TAO to outperform TBB by 3% and Qthreads by 4% on average. As with UTS, one might guess that *one-wide* assemblies should perform best in the first level of the reduction, given that flow-graph concurrency is ample and one-wide assemblies keep cores fully occupied. However, our tests show that one-wide assemblies are not as efficient as two-wide assemblies at this level. The reason can be found in the AMD Interlagos architecture (Figure 2), in which two adjacent cores share a 2MB L2 cache. At the first level of the reduction, the working set of one assembly is 2MB ($= 32768\,\text{integers} \times \frac{8\,\text{bytes}}{\text{integer}} \times \frac{4\,\text{sorts}}{\text{assembly}} \times 2\,\text{buffers}$). Two-wide assemblies effectively exploit L2 cache sharing by co-scheduling cooperating tasks. Selection of best assemblies is a complex tuning problem that should not involve the programmer. As part of our future work we want to analyze schemes that will perform automatic selection of assemblies without the assistance of the programmer.

Table 3 also shows the impact of using work stealing. We observe that performance is considerably better when no stealing is used. This occurs because random work stealing conflicts with the initial work placement strategy that we adopted when initializing the ready set of the flow-graph. Random steals can temporarily im-

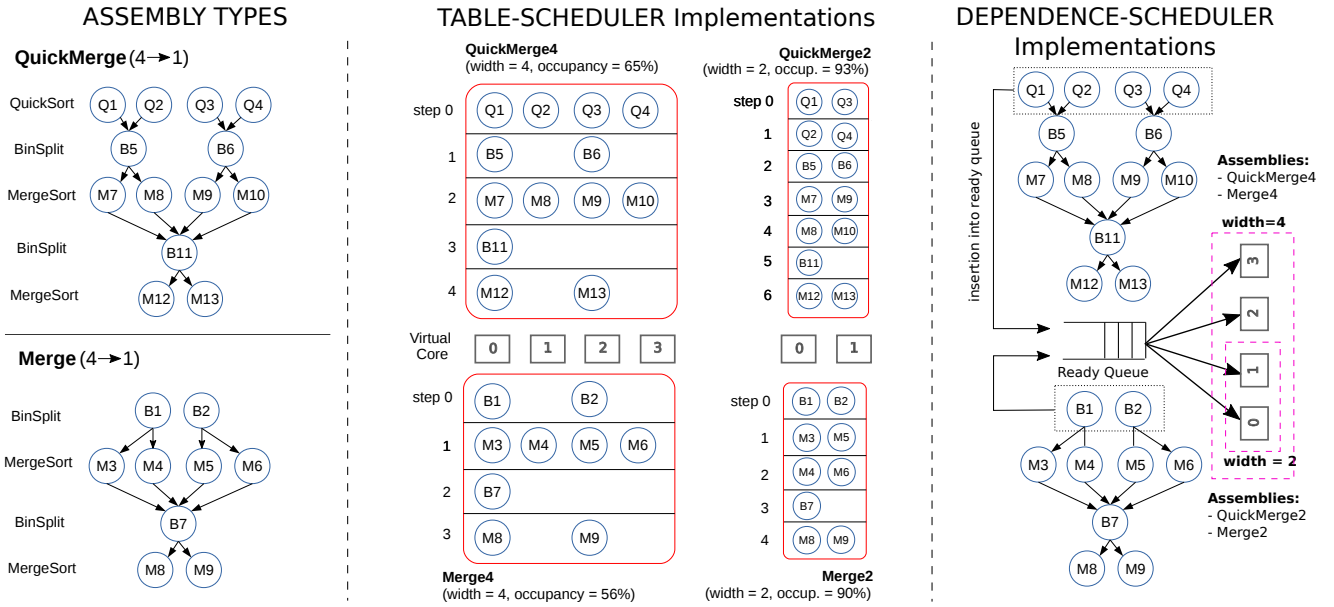


Figure 9. The Sort assembly library: The assembly functionality is shown on the left. `BinSplit` finds partitions to be sorted in parallel via `MergeSort`. Occupancy, shown for table assemblies, refers to the ratio of used slots vs. the number of total slots.

levels of width 2 + width 4		*1+4*	*2+3*	*3+2*	*4+1*
table	*w/ stealing*	6.87×	7.17×	7.38×	7.53×
table	*no stealing*	7.51×	**7.94×**	**8.07×**	**8.11×**
dependence	*w/ stealing*	7.11×	7.30×	7.46×	7.60×
dependence	*no stealing*	**7.81×**	7.93×	8.03×	8.06×
TBB		**7.87×**			
Qthreads		7.77×			

Table 3. Speed-ups compared to serial execution for hybrid TAO configurations, Qthreads and TBB. Upper levels in the reduction use *two-wide* assemblies, while lower levels are *four-wide*.

SYSTEM READ_ RESPONSES:	TBB	Qthreads	`go:TAO`
MODIFIED	5.639×10^7	1.005×	0.94162×
OWNED	5.641×10^4	1.3831×	0.68871×
SHARED	1.492×10^6	1.0684×	0.81633×
EXCLUSIVE	7.921×10^7	0.96805×	0.96525×
SUM	1.371×10^8	0.98425×	**0.9542×**

Table 4. Performance Counter results for Memory Traffic. Improvements are relative to TBB

prove load balance, but they eventually move working sets across NUMA nodes, which distorts locality and decreases performance.

6.3 Memory Traffic

We finalize by analyzing the impact of locality on memory traffic. We rely on AMD Interlagos' `SYSTEM_READ_RESPONSES` performance counter, which tracks the number of incoming cache lines transferred from the memory or interconnect (HyperTransport), and the cache lines obtained from other cores' data caches in the same NUMA node. Table 4 shows average counts for each component in the counter for the TBB runtime along with the reductions observed by the locality-aware Qthreads scheduler and by the `go:TAO` implementation. Overall, TAO shows the largest improvement, averaging at 4.5% reduction in memory traffic over TBB. Compared to TBB both Qthreads and TAO effectively reduce regular load misses (EXCLUSIVE). `go:TAO` also significantly reduces the lines transferred from another cache (SHARED) thanks to the L2 sharing within assemblies. This is a limitation of two-level schedulers such as Qthreads, which can address locality at the NUMA or L2 cache level, but not both. Furthermore, `go:TAO` also reduces the number of data cache store miss refills (MODIFIED) by executing assemblies atomically. These effects combined allow `go:TAO` to reduce the memory traffic by over 3% compared to Qthreads.

7. Related Work

Two level scheduling approaches are an active field of research since they can potentially provide good parallelism and locality. While hierarchical scheduling has been proposed at many different levels (e.g., cluster, rack, system, etc) our focus is on shared-memory systems.

One strategy is to implement hierarchy at the work queues. ForestGOMP [6] implements hierarchical queues associated to several levels of the architecture (NUMA, socket, core, etc). This can be used to constrain the execution of OpenMP teams to specific parts of the system architecture. Olivier et al. [17] proposed a hierarchical scheduler for OpenMP tasks which they implemented in the Qthreads runtime. Their approach groups the threads of a locality domain (such as a socket) into a *shepherd*, in which all threads share a LIFO work queue. This results in a per-socket schedule that is similar to parallel-depth-first [4], a provably efficient scheduler that exploits reuse on the serial (depth-first) path.

Several approaches have proposed to implement hierarchy in the work stealing algorithm. The SLAW scheduler [11] restricts work stealing to locality domains provided by the programming model such as Chapel's *locales* [7]. Min et al. propose HotSLAW [19], a hierarchical work stealer that extends SLAW with Hierarchical Victim Selection (HVS). HVS attempts to select a nearby victim worker to preserve locality and only selects increasingly non-local workers if no ready work is found.

go:TAO extends such hierarchical approaches by supporting both top-down and bottom-up specification of locality via *initial placements* [2], a topology-aware scheduler that selects closely connected sets of cores, and the internal locality encoded into the assemblies themselves. This allows to support complex memory hierarchies, as shown by the parallel sorting example in Section 6. Traditional approaches are mostly based on locality specification via annotations such as *places*. Such information alone is not enough to support multiple locality levels. Interestingly, in go:TAO the locality management is separated among different entities: 1) the global scheduler assigns topological partitions, 2) the library defines assemblies with high internal reuse, and 3) the programmer provides initial placements based on his privileged understanding of the application's data set.

We have showed how TAO can support mixed-mode parallelism. Wimmer et al. [23] developed an efficient runtime system for running tasks requiring multiple threads. In their approach, idle threads join teams to execute parallel regions. Their approach is lock-free, but it relies on a fixed thread hierarchy and furthermore requires all threads to join a team before executing the parallel section, two limitations that go:TAO relaxes. Recently, Sbirlea et al. have proposed "Elastic Tasks" [18], which are tasks that can be executed by variable number of processors. Similarly to Wimmer's approach, teams formation includes a waiting period. Elastic Tasks determine the threads dynamically based on system load. In a heavily loaded system an elastic task would obtain a single thread. While intuitive, our parallel sorting example shows that even for highly loaded systems choosing a single worker is not always the best option since it ignores working set size and the possibility of constructive cache sharing.

8. Conclusions

In this paper we have proposed Task Assembly Objects (TAO), an execution model targeting hierarchical many-core architectures. The main idea behind TAO is to match parallel work granularity to hardware topologies via a novel 2D computational units called task assembly. Assembly objects are nested parallel computations aggregating fine-grained tasks, cores and a private scheduler. A TAO computation is governed by a two-level global/private scheduler. The TAO prototype, go:TAO, takes a flow graph of assemblies and dynamically schedules assembly objects to partitions of the system topology. Our evaluation shows that go:TAO can quickly generate parallelism and exploit complex cache hierarchies via a combination of internal assembly reuse, initial work placements and topology-aware assembly mapping. Our experience with go:TAO suggests that a DAG of patterns can provide productive programming with TAO. As future work we plan to address automated selection of assemblies when multiple versions with different properties (width, sharing, etc) are available.

References

[1] Openmp application program interface. version 4.0, Jul 2013.

[2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12, 2000.

[3] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, et al. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[4] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, March 1999.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[6] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst. Structuring the execution of openmp applications for multicore architectures. In *2010 IEEE International Symposium on Parallel Distributed Processing*, IPDPS'10, pages 1–10, April 2010.

[7] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, Fall 2007.

[8] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press.

[9] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, 2009.

[10] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of SIGPLAN 1998*, June 1998.

[11] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of IPDPS'10*, May 2010.

[12] An Huynh, Douglas Thain, Miquel Pericàs, and Kenjiro Taura. Dagviz: A dag visualization tool for analyzing task-parallel program traces. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*, Nov 2015.

[13] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[14] Jun Nakashima and Kenjiro Taura. Massivethreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2014.

[15] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. Uts: An unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing*, volume 4382 of *Lecture Notes in Computer Science*, pages 235–250. Springer, 2007.

[16] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. Characterizing and Mitigating Work Time Inflation in Task Parallel Programs. In *Proceedings of SC12*, SC12, November 2012.

[17] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, Michael Spiegel, and Jan F. Prins. Openmp task scheduling strategies for multicore numa systems. *International Journal of High Performance Computing Applications*, 26(2):110–124, May 2012.

[18] Alina Sbirlea, Kunal Agrawal, and Vivek Sarkar. Elastic tasks: Unifying task parallelism and spmd parallelism with an adaptive runtime. In *Euro-Par 2015: Parallel Processing*, volume 9233 of *Lecture Notes in Computer Science*, pages 491–503. Springer, 2015.

[19] Katherine Yelick Seung-Jai Min, Costin Iancu. Hierarchical work stealing on manycore clusters. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS'11, Oct 2011.

[20] Nathan R. Tallent and John M. Mellor-Crummey. Effective Performance Measurement and Analysis of Multithreaded Applications. In *Proceedings of PPoPP'09*, February 2009.

[21] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Trans. Program. Lang. Syst.*, 36(3):10:1–10:51, September 2014.

[22] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[23] Martin Wimmer and Jesper Larsson Träff. Work-stealing for mixed-mode parallelism by deterministic team-building. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 105–116, 2011.