Elastic Places: An Adaptive Resource Manager for Scalable and Portable Performance

MIQUEL PERICÀS, Chalmers University of Technology

The diversity and complexity of modern computing platforms makes the development of high-performance software challenging. Designing *scalable* software requires tuning for a large set of resources, including cores (parallelism), memory bandwidths, and various levels of private and shared caches, as well as developing strategies for optimizing locality. But highly optimized implementations are often inefficient when executed on a different platform. This is the *performance portability* problem. One approach to scalability and portability is to tune the amount of work per task based on runtime overheads and concurrency. This results in a better balance between parallelism and scheduling overheads, but it can neither tune data reuse nor avoid inter-task interference. We propose a complementary approach that consists in tuning the amount of resources allocated to tasks and combine it with software-defined task topologies to provide portable locality. These ideas are combined into a low-overhead resource management scheme called *Elastic Places*. Elastic Places is implemented in the XiTAO software framework but the core ideas are equally applicable to other languages and runtimes. Experimental results on an AMD-based NUMA machine and an Intel Knights Landing system show that elastic places provides both high scalability and performance portability, with speed-ups of up to 2.3× on both platforms compared to state-of-the-art runtimes.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; • **General and reference** → **Performance**; • **Computing methodologies** → **Parallel programming languages**; • **Computer systems organization** → Multicore architectures;

Additional Key Words and Phrases: Dynamic scheduling, constructive sharing, parallel slackness, NUMA-aware scheduling

ACM Reference format:

Miquel Pericàs. 2018. Elastic Places: An Adaptive Resource Manager for Scalable and Portable Performance. *ACM Trans. Archit. Code Optim.* 15, 2, Article 19 (April 2018), 26 pages.
<https://doi.org/10.1145/3185458>

1 INTRODUCTION

To fully make use of all the cores in modern multicore processors, it is necessary to parallelize applications by decomposing them into concurrent tasks. But parallelism alone is not enough, as the scalability of parallel software quickly becomes limited by several system bottlenecks. Consider for example the scalability of a two-dimensional (2D) Jacobi stencil, as shown in Figure 1. The

This research was supported by the Swedish Foundation for Strategic Research (SSF) under grant RIT10-0033 and by the European Research Council under grant 340328.

Author's address: M. Pericàs, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden; email: miquelp@chalmers.se.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 1544-3566/2018/04-ART19 \$15.00

<https://doi.org/10.1145/3185458>

Fig. 1. Performance analysis of a greedily scheduled 2D stencil on the ABU (left) and KNL (right) systems.

Fig. 2. From classic task scheduling to XiTAO.

figure shows the effects of scaling a 2D Jacobi stencil computation on two modern platforms: (a) an eight-way NUMA system with 48 cores consisting of $4 \times$ AMD Abu Dhabi processors (ABU) and (b) a 64-core/256-threaded Intel Knights Landing processor (KNL). The stencil is implemented as a flow-graph using TBB [26]¹ and is decomposed by partitioning the 2D grid in the x and y axes to match the number of threads of each platform. The plots depict the total work time (= time in seconds spent executing the stencils), the *parallel idleness* [42] (= percentage time in which threads are not doing any work), and the speed-up compared to the serial execution. As more threads are added to the computation, several bottlenecks limit linear scalability, including (1) contention due to limited memory bandwidth, (2) decreasing parallel slackness (insufficient concurrency), and (3) interference from threads sharing a single core (multithreading). Good scalability is only possible given enough functional units, cache capacity, memory bandwidth, and concurrency.

Figure 1 exemplifies the difficulties of developing software that scales and runs efficiently on more than one platform. A careful balance between different kinds of resources needs to be found for each platform along with a task granularity that provides enough concurrency but does not unnecessarily burden the scheduler. Coarsening fine-grained tasks [15, 27, 32] can find a better balance between parallelism and scheduling overheads, but it does not address interference nor can it easily deal with contention on the memory bandwidth or the processor's caches. Instead, we propose to extend task coarsening with resource aggregation (i.e., matching coarse tasks to the required resources). Figure 2 describes the three conceptual steps from the classic 1-task to 1-thread method (1-to-1) to the M-to-N scheme that we propose in this work.

¹Flow-graphs are an implementation of task directed acyclic graphs (DAGs). Task DAGs eliminate unnecessary barriers and enable producer-consumer locality across iterations, one of the limitations of classic OpenMP implementations.

In this article, we propose an execution model and runtime to achieve scalability for applications based on task DAGs. The scheme is based on two main concepts. First, we propose to dynamically assign sets of resources to groups of tasks. This is achieved by executing tasks within *Elastic Places*. *Places* are resource containers in which groups of tasks execute [10, 11]. *Places* provide resource isolation, which enables controlled resource sharing and protection against interference. *Elasticity* refers to the dynamic aggregation and disaggregation of resources. We use elasticity to match places to task-level resource requirements. Second, we provide portable locality-aware scheduling by using *software topologies*. Software topologies are spaces, such as a 2D or 3D torus, which the programmer uses to assign locations for particular tasks. This provides a measure of distance between tasks that the runtime then uses to select appropriate places for locality-aware execution.

Elasticity may require adjusting the number of cores on which tasks execute. Hence, to exploit elasticity software needs to be moldable [19], which we achieve by leveraging dynamic scheduling. We propose to group multiple tasks into a *Task Assembly Object* (TAO) and annotate it with a resource requirement (e.g., a number of cores). The grouping of tasks into TAOs targets reduced inter-TAO communication. Intra-TAO communication is effectively addressed by the shared caches and high-speed on-chip networks common in today's multicore architectures. Each TAO is executed by its own embedded scheduler. This adds another degree of flexibility to the scheme. In essence, a TAO is similar to a nested parallel runtime requesting arbitrary resources.

We have implemented these strategies in our XiTAO² framework.³ This article describes the implementation of elastic places and software topologies in the XiTAO framework and evaluates its scalability and performance portability. In summary, we make the following contributions:

- We introduce elastic places, a novel method that achieves scalability and performance portability by dynamically setting the amount of resources provided to parallelized tasks.
- We introduce a portable locality interface based on software-defined topologies, which integrates naturally with elastic places.
- We describe a novel scheduler called *Dynamic Place Allocation* (DPA) to support elastic places. The scheduler dynamically allocates, maps, and frees hardware places. DPA has no centralized components and allows participating threads to proceed asynchronously.
- We describe XiTAO, a software framework that enables the description of moldable tasks with arbitrary resources and that implements the DPA scheduler and software topologies introduced by this work.

This work is organized as follows. Section 2 summarizes basic concepts on parallelism and communication. Section 3 describes how to achieve scalability and performance portability by leveraging elastic places. Section 4 describes our implementation of these concepts within the XiTAO framework. The experimental methodology is described in Section 5 and the XiTAO implementation of elastic places and software topologies is evaluated in Section 6. Section 7 covers related work and Section 8 summarizes our conclusions.

2 PARALLELISM AND COMMUNICATION

Writing parallel software requires identifying sections of code that can be executed in parallel. Classic approaches are mostly based on series-parallel computations [21, 30], in which parallel loops are interleaved with sequential sections. This model is simple to understand but the serial sections impose global synchronization that limit scalability. This has motivated research into schemes based on local synchronization and lazy generation of parallelism [20, 32]. Such

²Where Xi represents the greek letter ξ , which here stands for eXtreme scalability.

³<https://sites.google.com/site/mpericas/xitao>.

Fig. 3. Left: Summary of metrics that are used in this work. Right: Relation to system architecture.

computations are efficiently represented as a DAG of tasks, such that a node represents a single-threaded computation and an edge represents a precedence constraint, usually due to point-to-point communication. Intuitively, the more parallelism one attempts to exploit, the more communication that will be generated. This is a fundamental tradeoff in parallel computing that affects both data locality and scheduling overheads.

Historically, communication was cheap and scheduling focused on minimizing execution time by maximizing parallelism. Greedy schedulers are an example of this [7]. However, as communication costs have increased over the past decades, the focus has shifted towards finding a balance between parallelism and communication. Finding a good balance between parallelism and communication is still an active field of research despite many years of work [2, 4, 5, 8, 12–14, 18, 23, 31, 35, 44, 50]. One approach to reducing communication is to increase the granularity of tasks [15, 27, 32]. This reduces scheduling activity and forces more communication to happen within a single core, hence exploiting private cache hierarchies. Globally, a common theme is to assign a home (i.e., a *place*) to particular tasks. In early approaches places were single threads to which particular tasks were bound [2]. The intuition is that communicating tasks should run on the same cores to reduce inter-core communication. However, such schemes are too restrictive. First, the private L1/L2 caches in modern processors provide only a limited capacity for data reuse. And second, strict locality may lead to the overloading of particular cores while other cores sit idle. Since the shared caches of modern multicores offer higher capacity and service more cores, it is useful to extend the notion of place to represent the various hierarchical levels of caching provided by modern platforms. Most proposals in this direction are static—that is, they consider fixed hierarchical places [10, 11, 48]. Our notion of elastic places provides both dynamically generated places along with a mechanism to reduce scheduling overheads. In addition, global locality is provided by software topologies. These concepts are described in the following sections.

3 SCALABLE SCHEDULING OF DAGS

Modern platforms have many shared resources that need to be taken into account to achieve high performance. Figure 3 (left) summarizes several resources we consider in this work along with the metrics that characterize them. Figure 3 (right) shows their mapping to the system architecture, when applicable. In addition to these, we expect to add more resources in the future. One example is power sharing, which will become critical with the advent of dark silicon [17].

Ideally, a computation should operate in a regime of zero waste. This is a “balanced” situation in which a computation is not claiming resources that it cannot use effectively. Assuming enough

task parallelism, performance scales linearly with resources while in the zero-waste regime. This section describes how a DAG-based computation can improve scalability by aggregating and disaggregating resources.

In classic scheduling schemes, tasks are non-preemptive single-threaded computations with fixed entry and exit points. Let us assume that a task A can be described by an amount of work T_A (in seconds), a reuse distance curve (identifying also the working set size W_A) and a miss rate for cache level L of $MR_{A,L}$. Referring back to Table 3 (right), let us assume Task A is given access to cache $CC_{3,1}$ via core 1 of NUMA node 1. The resulting miss rate $MR_{A,3}$ will be negligible if $W_A < CC_{3,1}$, or a function of $W_A - CC_{3,1}$ otherwise. Assume now that the cache is shared among two cores (1 and 2). If a second task B with cache miss rate $MR_{B,3}$ is scheduled on the companion core 2, then the aggregate requested memory bandwidth will be $MR_{A,3} + MR_{B,3} + MRi_{A,B}$, where $MRi_{A,B}$ is the extra bandwidth resulting from destructive cache interference between tasks A and B . As long as the memory bandwidth MB_1 is large enough (i.e., $MB_1 > MR_{A,3} + MR_{B,3} + MRi_{A,B}$) the system can absorb the extra computation without saturating the DRAM channels. However, if $MR_{A,3} + MR_{B,3} + MRi_{A,B} > MB_1$, then the memory system will become saturated and execution will become wasteful. In the special case $W_A = CC_{3,1}$, task A will execute efficiently on core 1, but no other task should be scheduled on core 2. This explains why greedy schedulers are not efficient in this scenario. If ready tasks were available, then a greedy scheduler would schedule them to the idle core, creating interference with task A . To avoid this situation we would like Task A to get exclusive access to the shared cache $CC_{3,1}$ and, as a result, avoid other tasks from executing on core 2. Enforcing idleness on a core is potentially less wasteful than allowing interference between multiple threads. If two threads share the same data, then there is no waste and instead a higher cache efficiency can be achieved by co-scheduling them to the same cache.

Hence it is desirable to execute DAGs in which individual tasks can map from 1 to P cores and span different levels of caching. This yields a DAG-based model in which individual tasks can be expanded to match their internal resource requirements, a concept similar to *moldability* [19].

At the system level, the runtime needs to support this moldability by dynamically assigning appropriate hardware partitions that provide the necessary resources. Given that resources change from system to system and they furthermore depend on runtime conditions, such hardware places need to support *elasticity*—that is, it must be possible to dynamically aggregate and disaggregate resources. This is the motivation for *Elastic Places*.

The elasticity feature addresses the two remaining bottlenecks: local scheduling overheads and global parallel slackness. Local scheduling overheads can dominate if the inner work-sharing is too fine grained. For tasks of duration T_x and P cores, work-sharing using a single queue (a common implementation of OpenMP dynamic) becomes a bottleneck if the global scheduling time T_{gs} satisfies $T_{gs} > T_x/P$. Compared to work-sharing, work-stealing has better scalability. A bottleneck occurs only if the time for stealing T_{st} and the time for local scheduling T_{ls} is bigger than the task execution time T_x —that is, if $T_{ls} + \alpha \times T_{st} > T_x$, where α is the percentage of stolen tasks.

Another side effect of elasticity is on the parallel slackness of the DAG computation. High parallel slackness makes critical path scheduling less important in the selection of tasks. Furthermore, it allows the scheduler to choose among more ready tasks, which can be used to optimize inter-task locality. If a system has P processors, then high parallel slackness is achieved if the average parallelism of the computation ($\bar{P}$) satisfies $\bar{P} \gg P$. This constraint restricts traditional DAG models, in which the number of workers is the same as the number of physical processors in the system. The constraint is relaxed by using moldable tasks. If all tasks are given W cores, then the dimension of the system is reduced to only P/W places. Hence the new constraint becomes $\bar{P} \gg P/W$, which is often considerably easier to achieve.

4 THE XiTAO MODEL AND RUNTIME

4.1 Task Assemblies and Application DAGs

The concepts introduced in the previous section are implemented in a software framework called XiTAO. XiTAO implements a scheme in which fine-grained tasks are consolidated into work containers called *Task Assembly Objects* (TAOs). Each TAO is dynamically assigned a hardware resource partition at runtime. By dynamically scheduling the inner DAG computation, TAOs can be provided with different number of resources. This makes TAOs moldable. To support generalized parallel patterns beyond simple SPMD-like parallel loops, XiTAO delegates the inner scheduling to the TAOs themselves. Hence, from the perspective of XiTAO, a TAO is a black box requesting a certain amount of resources to execute. XiTAO is not tied to a particular programming model but is designed to serve as the back-end for the family of DAG-based programming models. The amount of resources requested by a TAO is specified as a resource hint that the runtime uses to find a suitable resource partition. The hint is a suggestion and can be overridden by the runtime system during execution, for example, when a TAO requests more resources than available in the system.

In summary, each Task Assembly Object (TAO) consists of three components:

- (1) *Work*: The work is represented by a set of tasks, possibly with dependencies. TAOs behave similarly to functions. Like functions, they can be configured via parameters, and they may change the memory state. TAOs cannot be preempted by other TAOs. Hence, when a TAO spawns new TAOs, a help-first [22] policy is applied that delays its execution until the parent has completed.
- (2) *Embedded scheduler*: The embedded (or *internal*) scheduler executes the TAO on a set of *virtual* cores. It is defined by the TAO itself and is thus fully general. Both static and dynamic schedules can be implemented, but should be dynamic to support moldability.
- (3) *Resource hint*: The hint suggests an amount of resources to be allocated by the runtime scheduler so that the TAO executes efficiently. The hint may refer to various kinds of resources such as cores, caches, or memory. In practice, programs always run on OS threads, hence the runtime translates resource requirements into core requirements. The resource hint can be static or set during execution based on parameters.

The full program is described as a dynamic DAG of TAOs, called the *TAO-DAG*. A DAG is a generic representation that allows XiTAO to serve as a back-end for many parallel computation models, including fully strict computations [7], async-finish parallelism [22], loop-based parallelism [1], and task-dataflow models [36]. The edges of this DAG indicate ordering dependencies between TAOs. The case of a TAO-DAG consisting of single-threaded TAOs is conceptually identical to a classic task DAG.

Figure 4 shows the architecture of XiTAO, highlighting the interactions between (a) the programming front-end, (b) a single TAO, (c) the TAO-DAG, (d) the runtime, and (e) the system architecture. The programming front-end is not part of the XiTAO framework but is shown here for completeness. The diagram showcases the execution of a 2D heat diffusion kernel and shows TAOs with different resource requirements (core-widths $w = 1$ and $w = 2$).

4.2 Design of Scalable XiTAO Computations

The XiTAO runtime executes programs provided by a programming front-end (Figure 4(a)). The front-end is not part of XiTAO. For completeness, we summarize the tasks that this front-end needs to perform to generate XiTAO computations.

Fig. 4. The components of the XiTAO framework, showing the execution of a heat diffusion kernel with TAOs of widths $w = 1$ and $w = 2$ on a small multicore machine with two NUMA nodes. The programming front-end is not part of XiTAO but is shown here for completeness.

Table 1. Steps for Generating a Resource-efficient XiTAO Computation Expressed as a TAO-DAG and TAO Objects with Associated Resource Widths

Step #1: Partition Task DAG into TAO-DAG	Goal 1: Minimize communication across TAOs. This goal targets <i>locality</i> .
	Goal 2: Achieve a TAO-DAG parallelism of at least the number of available cores. This goal targets <i>parallel slackness</i> . Note that the slackness will improve by using wide TAOs (i.e., $\overline{W}_{TAO} > 1$), since $\overline{P}_{TAO-DAG} > \frac{P_{system}}{\overline{W}_{TAO}}$.
Step #2: Characterize TAOs $\{A_i\}$ and assign resources	If(Compute Intensive): The goal is to minimize scheduling overheads. <ul style="list-style-type: none"> • For work-sharing: $T_{sched} \ll T_{exe}/P$; • For work-stealing: $T_{sched} + \alpha \times T_{st} \ll T_{exe}$; • For static schedules we assume negligible overheads
	If(Cache Intensive): The goal is to find the smallest cache subset a such that $WS_A \leq CC_a$
	If(Streaming Intensive): The goal is to find a number of cores M so that the BW of the NUMA node k (MB_k) is not saturated (i.e., $M = MB_k/MR_{A,LLC}$). The width of the TAO is then the number of cores per NUMA node N divided by this number (i.e., $\frac{N \times MR_{A,LLC}}{MB_k}$) rounded upwards to match the cache hierarchy.

We assume as starting point a conventional task DAG-based computation consisting of single-threaded tasks (the nodes) and dependencies (the edges). The goal is to design the TAOs and the TAO-DAG and assign resource requirements targeting efficiency and scalability. To determine the size and resources of a TAO, one prerequisite is to analyze the individual tasks of the DAG to understand their algorithmic intensity, working set size and reuse distance. Table 1 describes a procedure to systematically construct the XiTAO application. We summarize how to apply this procedure to the 2D stencil computation discussed earlier. The main components are described in Figure 4(b) and (c).

The heat diffusion benchmark computes a heat distribution on a 2D grid. To achieve this, one strategy is to perform a 2D Jacobi stencil (a type of nearest-neighbor computation) into a replicated 2D grid and subsequently copy the result back into the original 2D grid. The first step is to coarsen the stencil and copy operations into `jacobi2D` and `copy-back` TAOs such that only ghost points are communicated across TAOs (Goal #1). We stop coarsening when there is one TAO per core, which results in P_{system} TAOs (Goal #2). Figure 4 shows the resulting TAOs (part (b)) and TAO-DAG with the two types of TAOs (part (c)).

The global TAO-DAG is parallelized by decomposing the 2D grid in the x and y dimensions (Figure 4(c)). Similarly, the same partitioning is conducted internally to each TAO in order to match the resources of each hardware place (see Figure 4(b)). To set the resource requirements we need to understand the memory request rate of the stencil executing on a single core. If this rate is MR_A and the memory bandwidth of the NUMA node is MB_k , then MB_k/MR_A stencils can be executed in parallel before the NUMA node bandwidth is saturated. The resource width for each TAO should then be the number of cores per NUMA node N divided by the number of cores needed to saturate the bandwidth (i.e., $\frac{N \times MR_A}{MB_k}$). As a result, the higher the available bandwidth, the narrower the TAO should be. This is reasonable, since TAOs claiming less resources will enable more TAOs to be executed in parallel on the same resources, hence increasing performance.

For the purpose of proof-of-concept, in this article we construct the TAO-DAG and determine resource requirements by hand. Determining resource requirements is particularly challenging. Options to automatically detect resource requirements include profiling, prediction, and compilation. Profiling based schemes can be used to explore TAO configurations that minimize execution time over a period of time. The optimal values can then be used to configure the TAOs. *Predictive* schemes based on hardware performance counters can be used to estimate optimal resource requirements based on measurements of events. These models can then be used to set the best amount of resources. Finally, *compilers* can also derive the amount of required resources by analyzing details such as the number of memory operations and the number of arithmetic operations performed by a TAO.

4.3 Dynamic Place Allocation

Once the TAO-DAG and TAOs have been designed, we need a mechanism for dynamically allocating hardware places based on the resource requirements of TAOs. The place allocation algorithm should be fast and ensure dead-lock freedom. This requirement bears similarities with other techniques such as Deterministic Team Building [46], Elastic Tasks [41], and the hypervisor scheme of StarPU [24]. The main choices are between a centralized or a distributed scheme and between synchronous and asynchronous team join and exit. In XiTAO, a team refers to the set of all threads that form a place and are responsible for executing one TAO. In a centralized scheme, a single entity tracks the activity of all threads and decides which threads will join a new team. This is the approach taken by StarPU [24]. To achieve greater scalability, a distributed scheme in which the threads independently join teams is required. Deterministic Team Building and Elastic Tasks follow this approach. Both use a blocking scheme, in which threads wait until the team is formed or until a timer expires (case of Elastic Tasks). These waits introduce unnecessary synchronization that we would like to eliminate.

Figure 5(a) shows the design of the XiTAO DPA algorithm. In this figure, each color represents a unique elastic place within which a TAO is executed. The goal is to allocate a new hardware place with the required resources when a new TAO is scheduled. The hardware place becomes the location for the *future* execution of the TAO. Since XiTAO's hardware places are dynamically constructed partitions of the available resources we call them *Elastic Places*.

To implement the scheme we propose to insert pointers to the decoded TAO from a set of *Work Stealing Queues* (WSQ, see Section 4.4) into a set of *Assembly Queues* (AQ) collectively representing its hardware place. The TAO waits in the AQs until it is asynchronously fetched and executed by each worker thread. Figure 5(a) shows how places of different width can be interleaved. Deadlock is avoided by making the distribution of TAOs indivisible and by forcing FIFO ordering of TAOs in the Assembly Queues. An extrae/paraver [3] trace of an execution of this algorithm is shown in Figure 5(b), depicting a 48-core machine executing a quaternary reduction. The figure shows

Fig. 5. (a) DPA algorithm and (b) the DPA execution of a reduction over 48 cores. Each color is one hardware place. White is idle time. x-axis=time. y-axis=threads.

how the width of hardware places (i.e., the number of cores constituting the place) increases over time ($w = \{2, 6, 12\}$) to support larger cache capacity and bandwidth in each step of the reduction. An important characteristic of the DPA algorithm is its distributed nature, without any centralized components. We next describe the XiTAO prototype implementation targeting shared memory environments.

4.4 XiTAO Implementation

To evaluate the XiTAO execution model, we have developed a prototype runtime as a proof-of-concept with the goals of simplicity, portability, and efficiency. XiTAO is built on top of C++11 and makes use of C++11 threads and atomics.

The main task of the runtime scheduler is to allocate hardware places in a topology-aware way and to execute *ready* TAOs within these hardware places. The runtime uses distributed queues with work-stealing and has no centralized control (Figure 5, step 1). To provide a TAO with multiple cores XiTAO keeps two queues of ready objects per worker: a work stealing queue (WSQ) and an assembly queue (AQ). The per-thread WSQ is used for load-balancing ready TAOs. XiTAO follows the Cilk5 [20] stealing policy on top of this queue: (1) newly created TAOs are locally pushed and retrieved by each core using a LIFO policy and (2) global work stealing follows FIFO order.

When a worker thread fetches a TAO (Figure 5, step 2) it analyzes it and proceeds to distribute it by constructing an appropriate hardware place based on its resource requirement (Figure 5, step 3). The hardware partition is selected so that inner sharing is supported most efficiently. This is usually achieved by providing a shared cache and avoiding TAOs spanning more than one NUMA node. At this point the XiTAO runtime may override the resource requirement if the resource requirement cannot be currently provided. Distributing a TAO consists in enqueueing reference pointers into the AQs of each worker belonging to the place (Figure 5, step 4). Deadlock freedom is achieved by obtaining a lock for each of the participating AQs before inserting the pointers. The worker will always select a compact set of *assembly workers*, including itself, equal to the specified TAO width. The selected assembly workers represent the hardware place.

When a worker fetches a TAO from the AQ, it invokes the inner scheduler via the object's `execute()` method (Figure 5, step 5). The TAO-DAG scheduler sees the TAO execution as a black box. Worker threads enter and exit the TAO execution asynchronously, without entry nor exit barriers. If synchronization is required, then it needs to be performed by the embedded scheduler. This freedom allows TAOs to implement any kind of execution policies. When a TAO completes,

ALGORITHM 1: Simplified code of XiTAO's worker loop

```

1 int worker_loop(int _nthr){                               13 // 3. if so, execute it
2 // worker pinned to core #_nthr                             14 if(ptao){ ptao->execute(_nthr);
3 // AssemblyTask is a generic type for a TAO                15 if(ptao->completed()){
4 AssemblyTask *tmp, *ptao = nullptr;                        16 tmp = ptao->commit_and_wakeup(_nthr);
5 while(pending_tasks() != 0){                                  17 ptao->cleanup(); ptao = tmp; }
6 // 1. check forwarded path                                   18 }
7 if(ptao) { // construct elastic place                        19 // ptao may now contain one ready child
8 distribute(ptao, _nthr);                                    20 // 4. if not, check the local WSQ
9 ptao = nullptr; }                                           21 if(!ptao) ptao = check_WS_queue(_nthr);
10 // 2. check if a TAO is in the AQ                          22 // 5. if still not, try to steal one TAO
11 if(!ptao) ptao =                                           23 if(!ptao) ptao = steal_one_TAO(); }
12 check_assembly_queue(_nthr);                                24 }

```

at most one dependent TAO is scheduled immediately instead of being inserted into the WSQs. This is a common optimization to exploit producer-consumer locality between dependent tasks.

A low-level API for interfacing with XiTAO is described in Section 5. The code in Algorithm 1 outlines the implementation of the DPA algorithm (Figure 5(a)) in the XiTAO prototype. The distribution and execution of TAOs is shown respectively in lines 8 and 14. The throughput of the DPA algorithm is analyzed in Section 6.

4.5 Hardware Topologies

To map TAOs to hardware places with efficient inner sharing, XiTAO needs a view of the underlying system topology. This can be provided via a machine description file [40] or via a tool such as hwloc [9]. The current implementation of XiTAO uses a simpler approach that stores only the total number of cores and a table to help translate the cache hierarchy into the number of cores connected to each level. This scheme will be extended in the future to support more complex systems. XiTAO currently focuses on the following resources: cores (parallelism), caches (data locality) and bandwidth (communication).

TAO widths other than those corresponding to cores, caches or NUMA nodes, for instance, are not frequently used. However, we stress here that XiTAO uses a flat scheme that allows it to allocate hardware places of any arbitrary width in the range $[1 : \text{maxthreads}]$. As shown in Figure 5(a) and (b), hardware places can be disjoint, partially overlapping, or fully overlapping. To the best of our knowledge, such flexibility is not provided by any other approach.

4.6 Software Topologies

The scheme described so far focuses on providing locality *within* TAOs (i.e., inner sharing), while global load balancing is provided via work stealing. This is enough as long as inter-TAO communication is small and DRAM accesses are performed primarily within the local NUMA node. If either of these do not hold, then a scheme to ensure locality-aware producer-consumer mappings, NUMA-aware data placement or both becomes critical for performance. Take as an example the case of a streaming-intensive iterative 2D stencil computation, in which we decompose the 2D input domain in both dimensions (x, y) to create the TAO-DAG. To achieve good data reuse, each $\text{TAO}_i(x, y)$ should execute at the same cache as the $\text{TAO}_{i-1}(x, y)$, where i indicates the iteration number. Communication due to boundary points in the neighboring subdomains is small, hence reuse via the same cache structure is less critical. In addition to this, it is important that the TAOs of the first iteration are spread out to make good use of the system's resources and that the memory buffers are allocated in the corresponding NUMA node.

A mapping scheme should be general and support various forms of locality. In particular, we target a scheme that supports both data reuse via caches and data placement in memories. To achieve

Fig. 6. Applying software topologies to the iterative 2D stencil: The figure shows a subset of the TAO-DAG, with each TAO indexed by (x,y) . The arrow width indicates the amount of data communication between TAOs. For an $N \times M$ decomposition, the STA could be defined as $(x/N, y/M)$ to achieve a locality-aware mapping.

this we propose to overlay the TAO-DAG with a “virtual” *software topology*. This is achieved by assigning each TAO a location called a *software topology address* (STA) defined on a n -dimensional space with coordinates in the interval $[0,1)$. The parameter n is selected by the application. The XiTAO runtime then translates the STA into a particular hardware place and selects a WSQ into which the TAO will be inserted. Two TAOs sharing the same STA ($|STA_1 - STA_2| = 0$) are understood as having the highest degree of locality. The larger the distance $|STA_1 - STA_2|$, the lesser the two TAOs are assumed to communicate. When locality is high, XiTAO will attempt to map the TAOs to the same hardware place. This idea builds on topological mapping in MPI [43]. However, XiTAO’s software topologies are conceptually different. Our goal is to provide a measure of distance between two TAOs, not to indicate which sets of TAOs communicate, as the latter is already described by the TAO-DAG.

Figure 6 shows how the concept of STA can be applied to the iterative 2D stencil example. STAs offer a solution to bypass work stealing, but can also be used in combination with work stealing. XiTAO provides functionality to assign STAs and to copy the STA from one TAO to another. The latter is useful when the STA is not fixed, as can happen when work stealing is used. An experimental API for XiTAO is described later (see Algorithm 2).

An n -dimensional space is not a complete solution for locality. For example, it would be difficult to use this abstraction to describe the locality for irregular graph-based problems. Despite this drawback, its simplicity and its capacity to efficiently and effectively guide locality in many scenarios makes it an attractive option. The software topology scheme solves several problems. First, it provides portability as it depends only on the application but not on the platform. Second, by assigning an STA, both the problems of initial mapping of TAOs and of ensuring locality across TAOs are addressed. For communication-heavy scenarios such as a 2D Jacobi stencil, a guided task distribution (*push*) is more efficient than work-stealing (*pull*). Users of XiTAO can select which scheme to use based on application requirements. And third, software topologies provide a mechanism to support NUMA-aware data placement. This requires that data be allocated in memory by the first TAO that uses the data, or by an extra set of data-initialization TAOs. By exploiting software topologies and Linux’s first-touch policy together with local allocation (i.e., `numactl -l`), the scheme automatically allocates data in a NUMA-aware way.

5 EXPERIMENTAL SETUP

The goal of the evaluation is threefold. First and foremost, we analyze the potential of the XiTAO execution model to provide better scalability and portability. Second, we analyze the performance of the DPA scheduling algorithm. And, finally, we assess software topologies as a portable method for producer-consumer locality.

Fig. 7. Architecture of ABU (left) and KNL (right).

Table 2. Relationship between Cache Capacity and Hardware Place Sizes, Shown as [# cores (# NUMA)]

Capacity	16KB	32KB	1MB	2MB	4MB	8MB	16MB	32MB
ABU	1 (1)	2 (1)	2 (1)	6 (1)	6 (1)	12 (2)	24 (4)	48 (8)
KNL	1 (1)	1 (1)	2 (1)	4 (1)	8 (1)	16 (1)	32 (2)	64 (4)

5.1 Methodology

We utilize two platforms representing different design points in the multicore/manycore design spectrum. The first platform is a Dell PowerEdge R815 server with four AMD Opteron 6348 processors (Abu Dhabi microarchitecture), each consisting of two NUMA nodes and six cores per NUMA node, for a total of 8 NUMA nodes and 48 cores. The second platform is an Intel Xeon Phi 7210 processor (“Knights Landing”) configured as four sub-NUMA clusters (“SNC-4”) with 64 4-way multithreaded cores (total 256 threads) and the external MCDRAM configured in cache mode. We refer to these platforms as ABU and KNL, respectively. The platform architectures, cache hierarchies and cache capacities of both systems are shown diagrammatically in Figure 7. Table 2 shows the correspondence between aggregate cache sizes and hardware places in terms of cores and NUMA nodes. Although possible, hardware places that span more than one NUMA node need to be considered carefully. First, because inner sharing will involve costly cross-NUMA communication and result in data replication not present with shared caches. And, second, because the time complexity of allocating hardware places grows with the number of locks that need to be acquired, particularly across NUMA domains.

We evaluate the XiTAO implementations of several benchmarks and compare them with different state-of-the-art task-parallel runtime systems based on single-threaded (fine-grained) tasks. Intel Threading Building Blocks [26] serves as a baseline to which all benchmarks have been ported. Table 3 summarizes all configurations that are evaluated and shows the labels that are used in the evaluation section to identify each configuration. Qthreads (QTH) [45] and MassiveThreads (MTH) [33] are used when their particular features are relevant to the evaluation. All these runtimes schedule single tasks to individual threads and employ variations of work-stealing to balance the work. All the codes used in this work have been compiled with gcc 4.8.4 on ABU and with Intel icc 16.0.3 on KNL.

When measuring scalability, additional threads are mapped following a compact scheme—for instance, thread $\langle n \rangle + 1$ is mapped as close as possible to thread $\langle n \rangle$, except for multithreading.

Table 3. Evaluated Runtimes and Configurations, Highlighting Main Characteristics in Terms of Load Balancing, Granularity, and Selection of Ready Tasks for Execution

Config. (<i>Label</i>)	Elasticity	Load Balancing	Granularity	Task Scheduling
TBB (v4.4)	no	random work stealing	fine-grain	DAG or spawn-sync [6] + help-first [22]
Qth (v1.10)	no	places + inter-place stealing	fine-grain	spawn-sync with help-first
Mth (July 2016)	no	random work stealing	fine-grain	spawn-sync with work-first
XiTAO-wn-WS	n threads	random work stealing	coarse (TAO)	DAG (first ready + LIFO)
XiTAO-wn-topo	n threads	software topologies	coarse (TAO)	DAG (first ready + LIFO)
XiTAO-wA-topo	adaptive	software topologies	coarse (TAO)	DAG (first ready + LIFO)
XiTAO-wA-NUMA	adaptive	software topologies	coarse (TAO)	DAG + NUMA-aware allocation

When cores support multiple hardware contexts, the extra contexts are only allocated if the total number of threads requested by XiTAO is larger than the number of cores. This happens only with the KNL platform when using 128 and 256 threads. A similar consideration applies to memory allocation. In each run, threads are only allowed to allocate memory on the NUMA nodes on which they are executing. This is done to model different scales of hardware platforms, both in the number of cores and NUMA nodes (memory controllers). Most experiments use interleaved memory allocation via `numactl -i all`, which creates on average a flat view of the memory and increases bandwidth without requiring explicit data distribution. Considering a page-interleaved memory subsystem simplifies the analysis of results and is fairer, since the reference implementations do not implement NUMA-aware data placement. To demonstrate the effect of NUMA-aware placement, we also show its effects for the two memory intensive benchmarks (see configuration XiTAO-wA-NUMA).

Furthermore, we observe execution time variability due to randomized scheduling techniques, short duration of some benchmarks, and the number of threads, as more threads means more dynamic activity to distribute the workload. For example, in the case of Abu Dhabi, the worst case happens when the UTS benchmark is run on 48 threads. Here the randomness of work stealing and other system activity results in a worst case coefficient of variation (CV) of 25%. For benchmarks that do not use work stealing, the value of CV is around 5–7%. When the number of threads decreases, variability decreases also quickly. For example, UTS running on only six threads has a CV of $\sim 0.1\%$. To reduce variability we perform each experiment five times and report the median value. The generally small coefficient of variation plus the usage of a median of five experiments, clearly indicate that variability does not impact the validity of our results.

5.2 The XiTAO Low-Level Programming API

XiTAO is intended as a back-end for high-level models such as OpenMP or TBB, whose computations can be described as task DAGs and partitioned into a TAO-DAG. The low-level XiTAO API used in this work allows to explicitly describe TAOs and TAO-DAGs. The goal is to evaluate the performance features of the XiTAO model. Programmability is then achieved by using high-level programming models. For the same reason we did not implement automatic identification of resource widths following any of the methods outlined in Section 4.2. Instead, we set the TAO widths statically based on prior analysis of parallelism and working sets.

XiTAO provides low-level constructs to describe TAO objects as C++ classes and interconnect them into a TAO-DAG. Algorithm 2 shows simplified structures of the base `AssemblyTask` class, a minimal TAO class, and the basic XiTAO API. TAOs are specified as derived classes from the base `AssemblyTask` class. The `AssemblyTask` needs to be initialized with a resource width and,

ALGORITHM 2: TAO definition and XiTAO API

```

class AssemblyTask{ // defined by XiTAO
    int width; // resources := #cores
    int leader; // id of first worker in team

public:
    int set_sta(float X); // specify a STA (for a 1D topology)
    virtual int execute(int thread) = 0;
    int make_edge(AssemblyTask *); // create an edge
};

class TAO : public AssemblyTask { // defined by user or library
    TAO(args) : AssemblyTask(w){ // specify resource
        initialize(args);} // initialize TAO
    int cleanup(){...}; // for garbage collection
    int execute(int thread){ // invoke inner scheduler
        int tid=thread-leader; // obtain virtual id
        do_work(tid);} // invoke scheduler and start computation
};

// XiTAO runtime management API
int gotao_init(int numthreads=all); // initialize the runtime (default: full system)
int gotao_start(); // start a XiTAO computation
int gotao_wait(); // wait for a computation to complete
int gotao_push(AssemblyTask *); // push a ready TAO into a WSQ based on the VTL

// XiTAO TAO-DAG API
TAO *obj1 = new TAO(w,...); // crate a new TAO with resource width 'w' and parameters ...
// TAOs are automatically recycled by XiTAO after execution
obj1->make_edge(obj2); // add a dependency: obj2 depends on obj1
obj1->set_sta(0.5); // set a 1D STA (=0.5)
obj1->copy_sta(obj2); // copy the STA of a TAO

```

optionally, an address in the software topology, which in the shown example (bottom of Algorithm 2) corresponds to a 1D torus topology. TAOs can be created and pushed into the WSQs both before and after `gotao_start()` is called and execution begins. Dynamic generation of TAOs is critical to support irregular TAO-DAGs.

5.3 Benchmarks

Performance evaluation requires detailed analysis of parallelism, caching, and bandwidth for each individual case. Our analysis is based on a set of benchmarks representing different parallel patterns: *tree search*, *stencil*, *reduction*, and *N-Body*. We now describe their implementation in XiTAO along with the test inputs.

Unbalanced Tree Search (UTS). UTS [34] constructs a binomial tree in which each node has a probability q of generating m children. The UTS computation finishes when the full tree has been constructed. Each of the m children performs a short computation that reads data from the parent node. The working set is very small, so per-task cache resource requirements are small. To reduce communication due to parent-children data sharing, we assemble all children into one TAO. This results in a coarsened tree in which the newly spawned children of each node become a single child TAOs. Given that the UTS hash is still computed on a per-task basis, this change has no impact on the computation. Different TAOs then represent the parents and children in the UTS tree. By partitioning and coarsening the task graph, a large amount of parallelism can be generated quickly, further reducing load balancing overheads.

We compare against the MassiveThreads (MTH) and Intel TBB runtimes. MassiveThreads follows Cilk5's work-first policy [20] and is optimized for low-overhead stealing [33], an important feature for imbalanced workload such as UTS. TBB, on the other hand, uses the help-first policy and has slightly higher stealing costs. The TBB and MTH versions are based on a task-parallel generalization of the BOTS benchmark [16] developed by Huynh et al. [25]. The tested UTS input consists of a binomial tree with a probability $q = 0.05$ that each node has $m = 20$ children. The root branching factor is $b_0 = 100$, and the random seed is 806. This input results in a total of 1,471,121 nodes to be visited. We use the smallest compute granularity ($cg = 1$) to put maximum pressure on the schedulers. This results in sub-microsecond execution times for each task.

Heat Diffusion (HEAT). We introduced the HEAT benchmark and its XiTAO implementation in Section 4. The TBB implementation describes a flow-graph that is identical to the TAO-DAG of the XiTAO implementation. The XiTAO benchmark performs a decomposition following the cores and NUMA nodes of the platforms. The domain is decomposed in two dimensions $\{x, y\}$, both internally ($\{ix, iy\}$) and externally ($\{ex, ey\}$). The internal decomposition refers to the DAG inside the TAOs, while the external decomposition refers to the TAO-DAG. In the case of the ABU platform this results in an inner TAO decomposition of $\{ix, iy\} = \{6, 1\}$, enough to feed all combinations of 1 (L1), 2 (L2) and 6 cores (L3). In the case of KNL this results in an internal decomposition of $\{ix, iy\} = \{8, 1\}$, which supports places of widths 1 (L1), 2 (L2), 4 ($2 \times L2$), and 8 ($4 \times L2$). Decomposing only in the x -direction (column decomposition) reduces false sharing due to ghost points. We run HEAT for 500 iterations on an input grid consisting of $3,072 \times 3,072$ points. The input is again decomposed externally matching the platform topology: 8×6 partitions ($\{ex, ey\} = \{8, 6\}$) for the case of ABU and 32×8 partitions for the case of KNL. By making the TAO-DAG's concurrency identical to the platform's parallelism, we make the execution more sensitive to the scheduling. This extra pressure allows to better evaluate the performance of the tested runtimes. Each individual grid element has type `double`, leading to a working set size of 144MB, beyond the combined system's cache resources.

The XiTAO implementation of HEAT supports software topologies and NUMA-aware data allocation. These are achieved by attaching an STA to each TAO based on the position on the 2D grid. For the NUMA-aware allocation, we add an extra layer of copy2D TAOs before the computation starts. These TAOs allocate the buffers on the local NUMA node (assuming first touch policy with local allocation) and then copy the input data into the newly allocated buffers. Note that NUMA-aware allocation requires that the data touched by the local tasks falls in the same memory pages. For this reason, the NUMA-aware execution is only implemented for numbers of NUMA nodes that are a powers of two.

Parallel Sort (SORT). The parallel sorting code is based on the Cilk5 code originating from the Cilk5 [20] project, modified with a static cutoff for the mergesort recursion to generate a simpler reduction pattern. The TAO-DAG consists of a five-level reduction, as shown in Figure 8(a). To distribute the TAOs across the system, we assign a 1D STA following a similar strategy to HEAT. Figure 8(b) shows the two TAO *types* that compose the TAO-DAG and their inner scheduler: *QuickMerge* performs four Quicksorts and then merges them in parallel, while *Merge* merges four sorted arrays in parallel. The input consists of a fixed randomized array of 32×2^{20} 64-bit integers that leads to a working set of 512MB. The first level of the reduction decomposes the array into 256 TAOs, or total of 1,024 Quicksort tasks. The XiTAO versions are compared against TBB and QTH implementations that implement the fine-grained task graph in a spawn/sync programming style. We evaluate Qthreads, since its scheduling strategy (Multithreaded Shepherds [35]) is designed to improve locality, and sorting algorithms are cache-intensive. The XiTAO version of the code supports both software topologies and NUMA-aware data allocation. Both are achieved by setting initial STAs on a 1D topology based on the input data order (i.e., vector order). The

Fig. 8. (a) The TAO-DAG of the *Sort* benchmark and (b) the *Sort* TAO library: inner task graphs (left) and inner scheduler (right). Functions: BinSplit finds partitions then sorted in parallel via MergeSort.

dependent tasks do not use STAs and simply execute based on the general scheduling policy, which optimistically exploits locality by executing dependent tasks on the parent cores.

Fast Multipole Method (FMM). Finally, we evaluate an N-Body method based on the exaFMM code [49]. The input consists of 10^6 bodies, with up to 20 bodies per cell, and a multipole expansion order of 10. The main phase consists of a dual tree traversal (DTT) that traverses the destination and source octrees and constructs a vector of interactions for each cell to be processed. We partition the list of cells into a set of TAOs such that each worker thread processes about 400 cells. The interactions are scheduled using a dynamically computed static inner schedule such that each core processes blocks of 10 consecutive cells, assigned round-robin. We compare with the original implementation of exaFMM spawning one TBB task for each cell, and with an OpenMP-based parallel for solution that uses the default static scheduling.

6 EVALUATION

The evaluation focuses on the DPA implementation and the scalability and performance portability of the four benchmarks on the two test platforms.

6.1 Dynamic Place Allocation

We begin by analyzing the throughput of the DPA algorithm, as described in Sections 4.3 and 4.4. To this end, we have developed a microbenchmark designed to put the DPA scheduler under maximum stress. The microbenchmark generates and executes an array of TAOs containing zero work, hence all execution time is spent inside the DPA algorithm. We consider two scenarios and two variants of the place allocation algorithm. In the first scenario all TAOs are inserted into a single WSQ per hardware place. In the second scenario, the TAOs are spread and inserted from all WSQs. The two variants of the place allocation algorithm correspond to the asynchronous method described in Section 4.4 and to a blocking style that is designed to emulate alternative approaches such as Deterministic Team Building [46] and, to an extent, Elastic Tasks [41]. In this second variant, called *Barrier*, all assembly workers synchronize in a barrier before they proceed with execution. The microbenchmark measures the throughput at which hardware places of different width can be allocated from a single and from multiple queues. This metric can be used to measure both the efficiency of the XiTAO implementation as well as the cost of distributing a computation over several cores.

Figure 9 shows the measured throughput for both platforms. As expected, smaller TAO widths provide higher throughput. In both cases about 1×10^6 hardware places per second can be generated for TAOs of width 1 and 2. In general, allocating wider places is expected to decrease performance. The presence of a shared L2 cache in both platforms allows TAOs of width 2 to proceed at the same speed as TAOs of width 1, in particular for the case of barrier-free execution. We

Fig. 9. DPA throughput on the ABU (left) and KNL (right) platforms for both asynchronous style (default) and blocking style.

confirmed this by observing that other two-thread mappings not sharing an L2 do not achieve the same performance. As expected, the throughput for even wider places starts to decrease. The reason for this decrease is twofold: First, acquiring N locks to generate an N -wide place requires N cache line transfers, one for each core in a hardware place. Second, TAO pointers need to be inserted into N AQs (two transfers, one in each direction). Local access to the AQs by the assembly workers is lock-free and does not involve additional communication. Interestingly, the performance of the single queue and multiple queue schemes is almost identical. Only the ABU platform shows some divergence when more than two NUMA nodes are used. To allocate such large hardware places requires exchanging cache lines across different sockets. Using multiple WSQs achieves some overlapping of scheduling and execution and partially hides this latency. Blocking TAOs perform considerably worse than asynchronous TAOs, particularly in the case of the Intel KNL platform. In the worst case, throughput is almost one order of magnitude lower, which shows the benefits of asynchronous entry/exit. For the rest of the evaluation we will consider asynchronous TAOs only.

The performance of full-system TAOs is obviously quite low. Such TAOs are, however, rarely needed. In our research, we have not encountered any scenario in which having a TAO width higher than the width of a NUMA node offered better performance. This is reasonable, since there are no shared resources beyond a NUMA node. Hence, such a scenario results in considerable inter-NUMA communication.

6.2 Benchmarks

Figures 10 and 11 show the scalability results for the four benchmarks and two platforms ranging from 1 thread to the full system, including multithreading in the case of KNL (the shaded region in the plots). Overall, XiTAO achieves the best performance across all benchmarks, with up to 2.3× improvement in UTS for both platforms. In the following, we discuss the individual cases.

UTS. UTS generates many small tasks that makes it sensitive to scheduling overheads. Furthermore, each task has a small working set that fits in the L1. Given the small task lengths, data migrations to other L1 caches cannot be amortized. Since random work stealing does not consider parent–children locality, it steals tasks in a locality-unaware way, leading to performance degradation in TBB and MTH. The impact of the optimized work stealing in MTH is visible in the results. XiTAO addresses these two problems by (1) grouping all children nodes into a single TAO and (2) forcing all the nodes to exploit the parent–children locality in the same hardware place. UTS is designed to generate imbalance. By grouping multiple tasks into one TAO and mapping

Fig. 10. Scalability of benchmarks on ABU (top) and KNL (bottom) for UTS and HEAT. The multithreaded region of KNL is shown shaded. Labels are described in Table 3.

Fig. 11. Scalability of benchmarks on ABU (top) and KNL (bottom) for SORT and FMM. The multithreaded region of KNL is shown shaded. Labels are described in Table 3.

multiple resources at once, XiTAO can quickly adapt to imbalance. In the case of UTS, the working set of a TAO is very small and easily fits in the L1, making the benchmark compute intensive. Therefore, the best performing XiTAO configuration is XiTAO-w1-WS. We expect that a traditional runtime using static task coarsening could potentially achieve the same level of performance.

On KNL, performance degrades when multithreading is used. Concurrency is not ample in UTS that generates considerable parallel idleness. Idle threads are not harmless. They continuously

Table 4. Kernel Time (in Seconds) and *Parallel Idleness (%)* for *Heat*

Platform	ABU (48 Threads)			Platform	KNL (256 Threads)		
Configuration	jacobi2D	copy2D	idle	Configuration	jacobi2D	copy2D	idle
TBB	97.41	86.65	27.57%	TBB	90.68	68.60	55.5%
XiTAO-w1-WS	95.6	80.25	29.9%	XiTAO-w1-WS	84.06	26.41	91.69%
XiTAO-w1-topo	103.8	99.06	26.36%	XiTAO-w1-topo	92.89	60.84	27.01%
XiTAO-w2-topo	75.65	66.16	31.98%	XiTAO-w2-topo	91.75	54.74	31.35%
XiTAO-w6-topo	90.25	85.97	11.26%	XiTAO-w4-topo	91.11	52.70	33.58%

attempt to find work via work-stealing and by checking if work appeared in their local AQs. In a multithreading context, this activity interferes with the execution of actual kernels. Clearly, an efficient and scalable runtime needs a low cost mechanisms to find work when there are no ready tasks in the queues. Because work may suddenly appear in the AQs, setting worker threads to sleep is not a working solution in XiTAO. Currently, XiTAO implements a back-off scheme that exhibits decent performance, but researching better solutions to this problem will be necessary for future work. Note that, despite this problem, the XiTAO UTS implementations still outperform other runtimes by a considerable margin.

HEAT. The performance of HEAT scales well with the number of cores on both machines. To help understand the differences between the schemes we have instrumented the application with LOI-KRD [37]. LOI-KRD is a set of tools to measure parallelism, idleness, and reuse distance at the kernel level. Table 4 shows the total kernel time and idleness when the full systems are used (48 and 256 threads, for ABU and KNL, respectively).

On the ABU platform, we observe a trend toward decreased work time and idleness when the width of hardware places increases. The reduction in idleness is due to improved parallel slackness. As shown in Section 4, the runtime scheduler performs a greedy schedule on a platform having P/w schedulable places. As the value of W increases, the number of schedulable hardware places decreases. Consequently, parallel slackness increases, which reduces idleness and generates more locality-aware schedules by executing dependent TAOs back-to-back, thus exploiting producer-consumer locality. For XiTAO-w2-topo such a back-to-back execution is not possible, since dependencies force TAOs to execute essentially in lockstep: First, all the even TAOs execute, and then all the odd TAOs execute. Similarly, in the case XiTAO-w1-topo all TAOs are forced to execute side by side in full lockstep. Such lockstep executions become very sensitive to variability. We observed using *extrae/paraver* [3] that idleness is not uniform but follows a chaotic pattern with high idleness phases are followed by low idleness phases and vice versa. Configurations TBB and XiTAO-w1-WS behave similarly as both execute the same DAG and balance the load using work stealing. Given that most work is spawned locally, these implementations generate large amounts of work stealing activity, which result in high idleness and low performance.

To gain insight into the decrease in work time we generate reuse distance curves. Reuse distance gives an indication of the amount of data that is streamed between two accesses to the same variable. This correlates closely with temporal locality and cache efficiency. We rely again on the KRD [37] toolchain, which records per-thread kernel data access traces and generates aggregated per-socket reuse distance plots. Figure 12 shows the reuse distance curves observed at socket 0 (which includes NUMA nodes 0 and 1) for both SORT and HEAT. For the case of HEAT, the plots demonstrate how enforcing too strict scheduling via topologies (e.g., cases of XiTAO-w1-topo and XiTAO-w2-topo) can have a detrimental effect on short distance reuses. This happens because it

Fig. 12. Reuse distance curves measured at socket 0 of the AMD Abu Dhabi machine (NUMA nodes 0 and 1).

hinders back-to-back execution. This shows that topology-aware scheduling should be used only when there is enough parallel slackness, like the case of XiTAO-w6-topo.

The TBB implementation does not use topologies. It has good short distance reuse, because it is able to execute dependent TAOs back to back. However, its flat dynamic scheduler generates a lot of misses due to invalidations across sockets. At a reuse distance of 64MB, 19.1% of all misses are due to access to local data that has been invalidated by external writes (i.e., by other tasks). The large amount of invalidations explains why the hit rate remains almost constant when moving from 32MB to 64MB.

Figure 10 also shows the impact of using NUMA-aware data allocation. The effect is particularly significant for the case of HEAT. By eliminating almost all inter-socket data communication, performance scales linearly to the full size of the test system. At 48 cores (8 NUMA nodes), configuration XiTAO-w6-NUMA achieves a performance improvement of 45% on top of XiTAO-w6-topo.

On the KNL platform XiTAO's scalability follows a similar pattern, but the platform's external memory bandwidth to the MCDRAM is so large that the extra cache misses have a reduced impact on execution time. A bigger penalty is, however, observed in TBB at 128 and 256 threads and also for XiTAO-w1-WS. The results in Table 4 indicate that the lower performance originates from higher idleness. As with the ABU platform, at ≥ 128 threads, execution becomes very tightly synchronized due to the low parallel slackness, and as a result execution time becomes very sensitive to imbalance.

The XiTAO-w1-WS configuration behaves similar to TBB but already suffers reduced performance at 64 threads. This is explained by its sensitivity to the work stealing implementation, which uses a spinlock to protect each WSQ. When parallel slackness is low and software topologies are not used, load balancing is achieved by stealing most of the tasks. In this particular case, 98% of TAOs are stolen when running on 64 threads. As a consequence, idleness increases significantly at high thread counts. TBB uses a scheduling algorithm similar to XiTAO-w1-WS, but its lock-free implementation of work stealing scales better, which explains why TBB achieves better performance at 64 threads. For higher thread counts both implementations suffer due to lack of parallel slackness.

SORT. The SORT benchmark follows a quaternary reduction tree pattern with five levels exhibiting decreasing concurrency of 256, 64, 16, 4, and 1 TAOs at each level, corresponding to 1,024, 256, 64, 16, and 4 tasks for the TBB and QTH implementations. In this scenario, we test an adaptive XiTAO configuration in which the width of the hardware places increases as the reduction progresses. This scenario is particularly relevant, since the dynamic generation of places is a feature that only XiTAO supports. In the context of SORT, setting places adaptively is beneficial for two

reasons: In the first levels, the computation is dominated by caching requirements. Wider places provide more caching capacity. In the lower levels, the working set of each TAO becomes larger than the last-level cache capacity, and hence these TAOs have a streaming behavior. Allocating wide TAOs is beneficial, because additional cores can make better use of the available memory bandwidth. For the case of ABU, the chosen configuration executes the first three levels using TAOs of width 2 (to allocate L2 caches) and the bottom two levels use TAOs of width 6 (to allocate a full NUMA node). For the case of KNL the selected configuration executes the level zero using TAOs of width 2 (L2 cache), the levels one and two using TAOs of width 4 (the maximum parallelism of the TAOs), and the two lower merge steps using TAOs of width 16 (the width of one NUMA node in the SNC-4 configuration). Figure 8(a) shows the TAO-DAG, including the different levels of the reduction.

We begin by discussing the ABU results. At level three the working set size of each TAO is 128MB (64MB sort array + 64MB temporal buffer), which surpasses the per NUMA-node cache capacity. At this point performance becomes dominated by external memory bandwidth. The fact that XiTAO-w2-topo's performance is lower than XiTAO-wA-topo indicates that two cores are not enough to saturate the system's memory bandwidth. Using TAOs of width six better exploits the available bandwidth, since up to four Mergesort tasks can be executed in parallel (see Figure 8(b)). We also found that at the first level, using TAOs of width two is better than using a single core. At this level, the working set of one TAO is 2MB ($= 32768 \text{ integers} \times \frac{8 \text{ bytes}}{\text{integer}} \times \frac{4 \text{ sorts}}{\text{assembly}} \times 2 \text{ buffers}$), which matches the size of the shared L2 cache. Using TAOs of width two avoids interfering tasks from polluting the cache, which would occur if a greedy scheduler was used together with single-threaded tasks. This result contradicts conventional wisdom of greedy scheduling, since TAOs of width 2 enforce a certain amount of idleness, as internally the TAO has some sections with a parallelism of only one. This result demonstrates how performance sometimes requires to keep cores idle to avoid interference even when ready tasks are available. XiTAO blocks the resource and avoids conflicting accesses to the cache, which provides constructive cache sharing within the L2 cache while avoiding external interference.

To understand why the XiTAO configurations perform better than TBB and QTH we again instrument the benchmark with the LOI-KRD toolset and make use of the reuse distance analysis component [37]. Figure 12(b) shows the reuse distance curve for the SORT benchmark on ABU. The usage of software topologies enables the scheduler to perform an intelligent distribution of work, which avoids invalidation misses and leads to fewer cold misses compared to QTH and TBB. At a reuse distance of 16MB (similarly to the NUMA node's cache capacity), the XiTAO-wA-topo configuration has a lower percentage of distant reuses (17.9% lower than TBB and 8.7% lower than QTH), which explains the better performance of this memory-intensive benchmark.

XiTAO is also fastest on the KNL machine but the results show higher dependence on machine configuration, leading different configurations to perform best depending on the number of cores that are being used. Different bottlenecks apply depending on whether multithreading is active or not. This result suggests that future versions of XiTAO should implement a mechanism with adaptive resource requirements. On the KNL platform, particularly when multithreading is used, it makes sense to allocate a large number of cores for the lower levels of the reduction to exploit the bandwidth of the cores, hence XiTAO-wA-topo (with 4-wide and 16-wide TAOs) performs best at this level. However, such a big configuration is not efficient when the number of cores is low. That is because the TAOs feature only low internal parallelism and this scenario constrains the global (inter-TAO) parallelism too much. By exploiting the elasticity of the model, XiTAO can schedule appropriate per-TAO hardware places and thus adapt to the new scenario. In combination, these features enable good scalability and performance portability.

FMM. Compared to the other benchmarks, the performance improvement for exaFMM are modest. Instrumentation of FMM with LOI-KRD reveals that this code has very good data reuse and makes efficient use of caches. Inter-NUMA locality is less of a bottleneck, instead performance is mainly dominated by parallelism. In this context, using XiTAO can be beneficial as the embedded scheduling reduces overall task overheads. This argument is similar to UTS, with the difference that individual FMM tasks have larger granularity. Despite this, the scheduling overhead is not negligible. In the case of KNL, XiTAO achieves a 10% speed-up compared to TBB when scheduling over the full system.

Summary. The results highlight how elastic places improve both scalability and portability, achieving Speed-ups of up to 2.3× for the evaluation benchmarks and test platforms. As Figure 2 shows, XiTAO combines both coarsening and elasticity. Coarsening alone provides benefits to classical schedulers, as shown by configuration XiTAO-w1-WS. However, this often leads to cache and bandwidth overcommitting and therefore is beneficial mainly to compute-intensive benchmarks, as is the case of UTS. To balance the caching and memory bandwidth requirements, elasticity becomes a critical feature. Elastic Places provide resource isolation and achieves better parallel slackness, which is exploited in HEAT and SORT. The results show that elasticity is beneficial in two ways. First, it allows benchmarks to evolve their hardware places depending on the resource requirements of tasks, a feature that we exploited in SORT. And, second, elasticity simplifies portability. Both HEAT and SORT achieve good performance portability by adapting the sizes of hardware places on the two evaluation platforms. For other benchmarks that are neither compute intensive nor memory intensive, such as FMM, we found that elasticity improves performance when executing the same benchmark on different platforms, as hardware places provide an abstraction to manage resources across platforms, such as cores, caches, and memory bandwidth.

The configuration space of XiTAO is huge. However, the algorithm described in Section 4.2 shows how good configurations can be easily identified with a systematic procedure.

7 RELATED WORK

Scheduling parts of computations into fixed resource containers has been explored in the context of locality APIs [2] and scheduling heuristics that try to match nested-parallel computations on NUMA topologies [35]. The latter has been integrated into the programming model, as seen in X10 [11] or Chapel [10], which provide an abstraction for hardware *places*. However, their notion of places is static throughout program execution and they lack a mechanism to indicate the resources required by individual tasks. This article instead proposes to explicitly identify resource requirements and to dynamically match them to the hardware platform. Our evaluation shows the importance of providing such information to the runtime system.

XiTAO leverages several well-known concepts in locality-aware scheduling. Developing locality-aware schemes has been a long-running goal for dynamic task schedulers [2, 5, 35, 36]. Programming models such as Sequoia [18] allow programmers to annotate tasks with information about the memory hierarchy to improve performance portability and facilitate mapping. Legion [4] supports a more general set of access modes to *logical* regions, including concurrent access to shared read-only data. In addition Legion also provides a mapping interface to match machine details such as memory hierarchies. Sequoia and Legion provide good support for memory management, but they lack mechanisms to enforce constructive cache sharing among tasks. Furthermore, their single-threaded task nature makes efficient management of fine-grained parallelism difficult.

In this work, we particularly focus on enforcing constructive sharing of resources in parallel applications. Shared resource contention is an active topic of research both in software [50] and hardware [28, 38]. However, these works have focused only on multiprogrammed workloads consisting of single-threaded applications. Such workloads are far less challenging from a sharing

perspective, since they do not need to deal with issues such as constructive sharing or task scheduling overheads that motivated this work. One scheduler that targets constructive sharing is the parallel-depth-first (PDF) scheduler [5, 13]. Qthreads includes an implementation of PDF targeting multiple-socket machines [35]. Supporting constructive sharing usually requires decomposing a small (cache-sized) subproblem into multiple threads, often leading to fine-grained parallelism and non-negligible scheduling overheads. Proposals such as Carbon [29] or ADM [39] provide support for fine-grain parallelism by introducing special-purpose scheduling hardware.

Hierarchical runtime systems that support multi-threaded tasks have been proposed by Wimmer et al. [46] and by Sbirlea et al. [41]. Wimmer developed an efficient runtime scheme called *Deterministic Team Building* for running tasks consisting of data-parallel constructs. To provide efficient team generation, Wimmer's approach assigns cores following a fixed hierarchy. This contrasts with XiTAO's *flat* scheme, which supports arbitrary system partitions. Sbirlea et al. [41] proposed *Elastic Tasks* which can use a variable amount of cores to execute SPMD computations. Both Elastic Tasks and Wimmer's approach require a waiting time for team construction. XiTAO's DPA approach avoids any waits and allows threads to enter/exit TAOs asynchronously. While Elastic Tasks and Deterministic Team Building are designed to efficiently execute data parallel constructs, XiTAO is concerned with general DAGs and targets both scalability and portability.

Neither Deterministic Team Building, Elastic Tasks nor XiTAO support the automatic construction of task groups, instead relying on the programmer or compiler to generate the hierarchical task graph. Xia et al. [47] propose Dynamic Thread Grouping (DTG), one of the few schemes to support both automated grouping and automatic hardware partitioning. To achieve this, Xie et al. rely on a DAG described as a vector of tasks with dependencies, which is then partitioned at runtime. However, the scheme imposes several restrictions to support dynamic thread grouping. First, the full task DAG needs to be available and stored as a single list, which disables conditional parallelism and likely limits scalability. Second, DTG only works for homogeneous multicores and it supports only thread partitions of sizes 2^n , hence requiring the hardware to match these sizes. Unlike XiTAO, neither arbitrarily sized nor partially overlapping thread groups are supported. Finally, the Dynamic Thread Grouping algorithm considers only the tradeoff between parallelism and overhead, and it does not consider data reuse or sharing. These requirements make this proposal very difficult to consider in the context of XiTAO. However, this research points towards an interesting direction to consider in our future work.

8 CONCLUSIONS AND FUTURE WORK

Scalability and performance portability are major concerns in the current computing landscape characterized by large platform diversity. In this article, we have proposed elastic places, a resource abstraction that, combined with software topologies, provides scalability and portability to applications based on task DAGs. The main conclusions of this work are twofold. First, by coarsening tasks into work units that can be scheduled to arbitrary hardware partitions it is possible to achieve high scalability. Second, by adjusting resource requirements across hardware platforms we can achieve performance portability. These features have been added to our prototype runtime XiTAO and are supported by a scalable elastic places scheduler called DPA. Our experimental evaluation on multi- and many-core platforms resulted in Speed-ups of up to 2.3× compared to classic single-threaded runtimes. This shows great potential to address future computing systems. Our current work covers three directions. First, we are developing tools to construct XiTAO programs automatically from high-level languages. Second, we are developing the necessary technologies to automatically set resource requirements, which will further reduce developer intervention and ease portability. Third, we want to make the operating system aware of elasticity to enable

cooperation between the runtime and the OS and to avoid destructive interference between processes. Overall, this will lead to better QoS on the target machines.

ACKNOWLEDGMENTS

The author thanks Pedro Trancoso, Rio Yokota, and the anonymous referees for their valuable comments and helpful suggestions. Rio Yokota kindly provided access to the Intel Knights Landing platform used in this research.

REFERENCES

- [1] OpenMP. 2015. OpenMP Application Program Interface. Version 4.5. Available at <http://openmp.org/specifications>.
- [2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The data locality of work stealing. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*. 1–12.
- [3] Barcelona Supercomputing Center. 2013. *Extrae User Guide Manual*. Barcelona Supercomputing Center.
- [4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE, Los Alamitos, CA, Article 66, 11 pages.
- [5] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1999. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM* 46, 2, 281–321.
- [6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*. ACM, New York, NY.
- [7] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5, 720–748.
- [8] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst. 2010. Structuring the execution of OpenMP applications for multicore architectures. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS'10)*. 1–10.
- [9] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. Hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'10)*. IEEE, Los Alamitos, CA, 180–186.
- [10] B. L. Chamberlain, D. Callahan, and H. P. Zima. 2007. Parallel programmability and the chapel language. *Int. J. High Perf. Comput. Appl.* 21, 3, 291–312.
- [11] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. ACM, New York, NY, 519–538.
- [12] Quan Chen, Minyi Guo, and Zhiyi Huang. 2012. CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, NY, 163–172.
- [13] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. 2007. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*.
- [14] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. 2014. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Trans. Archit. Code Optim.* 11, 3, Article 30, 25 pages.
- [15] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. 2008. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*. IEEE, Los Alamitos, CA, Article 36, 11 pages.
- [16] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. 2009. 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)*. 124–131.
- [17] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2013. Power challenges may end the multicore era. *Commun. ACM* 56, 2, 93–102.
- [18] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*. ACM, New York, NY, Article 83.

- [19] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. 1997. Theory and practice in parallel job scheduling. In *Proceedings of the Conference on Job Scheduling Strategies for Parallel Processing (IPPS'97)*. 1–34.
- [20] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM Special Interest Group on Programming Languages Conference (SIGPLAN'98)*.
- [21] J. A. Gosden. 1966. Explicit parallel processing description and control in programs for multi- and uni-processor computers. In *Proceedings of the Fall Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS'66)*. ACM, New York, NY, 651–660.
- [22] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS'09)*.
- [23] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS'10)*.
- [24] A. E. Hugo, A. Guermouche, P. A. Wacrenier, and R. Namyst. 2014. A runtime approach to dynamic resource allocation for sparse direct solvers. In *Proceedings of the 2014 43rd International Conference on Parallel Processing*. 481–490.
- [25] An Huynh, Douglas Thain, Miquel Pericàs, and Kenjiro Taura. 2015. DAGViz: A DAG visualization tool for analyzing task-parallel program traces. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*.
- [26] Intel. 2017. Intel Threading Building Blocks. (2017). Retrieved May 18, 2017, from <https://www.threadingbuildingblocks.org/>.
- [27] Shintaro Iwasaki and Kenjiro Taura. 2016. A static cut-off for task parallel programs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT'16)*. ACM, New York, NY, 139–150.
- [28] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. 2007. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. ACM, New York, NY, 25–36.
- [29] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA'07)*.
- [30] Leslie Lamport. 1974. The parallel execution of DO loops. *Commun. ACM* 17, 2, 83–93.
- [31] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2014. Optimizing data locality for fork/join programs using constrained work stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'14)*. IEEE, Los Alamitos, CA, 857–868.
- [32] Eric Mohr, David A. Kranz, and Robert H. Halstead. 1991. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.* 2, 3, 264–280.
- [33] Jun Nakashima and Kenjiro Taura. 2014. MassiveThreads: A thread library for high productivity languages. *Concurr. Objects Beyond* 8665, 222–238.
- [34] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2007. UTS: An unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, Vol. 4382. Springer, 235–250.
- [35] 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 Perf. Comput. Appl.* 26, 2, 110–124.
- [36] J. M. Perez, R. M. Badia, and J. Labarta. 2008. A dependency-aware task-based programming environment for multicore architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing (Cluster'08)*. 142–151.
- [37] Miquel Pericàs, Kenjiro Taura, and Satoshi Matsuoka. 2014. Scalable analysis of multicore data reuse and sharing. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS'14)*. ACM, New York, NY, 353–362.
- [38] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'39)*. IEEE, Los Alamitos, CA, 423–432.
- [39] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible architectural support for fine-grain scheduling. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*.
- [40] Martin Sandrieser, Siegfried Benkner, and Sabri Pllana. 2012. Using explicit platform descriptions to support programming of heterogeneous many-core systems. *Parallel Comput.* 38, 1–2, 52–65.
- [41] Alina Sbirlea, Kunal Agrawal, and Vivek Sarkar. 2015. Elastic tasks: Unifying task parallelism and SPMD parallelism with an adaptive runtime. In *Euro-Par 2015: Parallel Processing*. Lecture Notes in Computer Science, Vol. 9233. Springer, 491–503.

- [42] Nathan R. Tallent and John M. Mellor-Crummey. 2009. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM, New York, NY, 229–240.
- [43] Jesper Larsson Träff. 2002. Implementing the MPI process topology mechanism. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*. IEEE, Los Alamitos, CA, 1–14. <http://dl.acm.org/citation.cfm?id=762761.762767>
- [44] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericas. 2017. Trends in data locality abstractions for HPC systems. *IEEE Trans. Parallel Distrib. Syst.* 28, 10 (2017), 3007–3020.
- [45] Kyle Wheeler, Richard Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the Workshop on Multithreaded Architectures and Applications (MTAAP'08)*.
- [46] Martin Wimmer and Jesper Larsson Träff. 2011. Work-stealing for mixed-mode parallelism by deterministic team-building. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. 105–116.
- [47] Yinglong Xia, Viktor K. Prasanna, and James Li. 2010. *Hierarchical Scheduling of DAG Structured Computations on Manycore Processors with Dynamic Thread Grouping*. Springer, Berlin, Germany, 154–174.
- [48] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2010. Languages and compilers for parallel computing. In *Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement*. Springer, Berlin, Germany, 172–187.
- [49] Rio Yokota. 2017. exaFMM. (2017). Retrieved May 18, 2017, from <https://github.com/exafmm/exafmm/>.
- [50] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 129–142.

Received July 2017; revised December 2017; accepted February 2018