GLT: A Unified API for Lightweight Thread Libraries

Adrián Castelló[1], Sangmin Seo[2], Rafael Mayo[1],
Pavan Balaji[2], Enrique S. Quintana-Ortí[1], and Antonio J. Peña[3]

[1] Universitat Jaume I de Castelló
{adcastel,mayo,quintana}@uji.es
[2] Argonne National Laboratory
{sseo,balaji}@anl.gov
[3] Barcelona Supercomputing Center (BSC)
antonio.pena@bsc.es

Abstract. In recent years, several lightweight thread (LWT) libraries have emerged to tackle exascale challenges. These offer programming models (PMs) based on user-level threads and incorporate their own lightweight mechanisms. However, each library proposes its own PM, exposing different semantics and hindering portability.
To address this drawback, we have designed Generic Lightweight Thread (GLT), an application programming interface that frames the functionality of the most popular LWT libraries for high-performance computing under a single PM. We implement GLT on top of Argobots, MassiveThreads, and Qthreads. We provide GLT as a dynamic library, as well as in the form of a static version based on macro preprocessing resolution to reduce overhead. This paper discusses the GLT PM and demonstrates its minimal performance impact.

1 Introduction

The number of processors in high-performance computing (HPC) systems has been continuously increasing, as reflected in the supercomputers of the June Top500 lists [5]. Following this trend, exascale systems are expected to leverage hundreds of millions of cores. Hence, future applications will have to accommodate massive concurrency.

Leveraging this massive intranode parallelism efficiently with traditional threading approaches may be difficult because of their relatively expensive context switching and synchronization mechanisms. In response, dynamic scheduling and lightweight thread (LWT) and tasklet models are designed to deal with the required levels of parallelism.

Different user-level thread (ULT) and tasklet libraries have been implemented in the past, such as Windows Fibers [14], Solaris Threads [2], ConverseThreads [13], Nanos++ [8], MassiveThreads [15], Qthreads [20], and Argobots [16] were the last three LWT solutions are compared. These solutions demonstrate semantic and performance benefits over the classic POSIX threads [3]. The variety of

The final publication is available at Springer via https://link.springer.com/chapter/10.1007/978-3-319-64203-1_34

LWT libraries, however, hinders portability. Their programming models (PMs) and internal strategies differ among implementations, and hence developing and maintaining applications and runtime systems for different LWT approaches require considerable effort. In this scenario, a unified standard interface can be highly beneficial, as long as it supports most of the functionalities offered by the LWT libraries while maintaining their performance.

In this paper we introduce the design of a unified LWT application programming interface (API), named Generic Lightweight Thread (GLT), that groups the functionality of popular LWT solutions for HPC under the same PM. To the best of our knowledge, this is the first paper proposing a unified API for LWT solutions oriented to HPC. GLT is presented as a proof of concept in order to spark a joint effort from the community to design a standard LWT API.

We implement GLT on top of Argobots, MassiveThreads, and Qthreads. The library choices are based on the work presented in [9], where a set of LWT implementations was reviewed, from the semantic point of view, using a set of OpenMP microbenchmarks.

In addition to a dynamic GLT library that enables switching the underlying LWT implementation, we provide a static version to minimize the overhead. Using the GLT API, application programmers can develop a single code for different LWT approaches. The design of a single API to take advantage of the functionality of different LWT libraries, along with an efficient implementation composed primarily of wrappers resolved at compile time, provides a semantically powerful, efficient framework for LWT programming.

Our experiments demonstrate the feasibility of a GLT implementation, which does not exert any perceivable negative performance impact on applications. In our experiments, the average performance overhead when using static and dynamic GLT approaches, instead of the original LWT libraries, is 0.08% and 0.6%, respectively.

In summary, the contributions of this paper are as follows: (1) analysis of the semantics/PMs of the three major LWT solutions for HPC; (2) design of a generic LWT API capable of offering the functionality of its underlying libraries efficiently; (3) practical demonstration of the GLT portability; and (4) experimental performance evaluation of the GLT API on top of three reference LWT libraries for HPC.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 offers background on our reference LWT libraries. Section 4 justifies the need for a unified LWT API. Section 5 discusses the GLT PM. Section 6 introduces the GLT unified API. Section 7 provides an in-depth performance analysis. Section 8 contains conclusions and future work proposals.

2 Related Work

In computer science, libraries commonly offer similar functionality. This situation may be caused by several circumstances, for example, a topic that is being developed by different institutions at the same time (e.g., MPICH [4] and Open

MPI [12]) or new implementations that aim to improve legacy or commercial codes (e.g., BLIS [19]).

In the past, some efforts have been made to join several solutions under a unique API. These common APIs aim to gather significant common features of the original libraries and offer them to users, who benefit from having to learn only a single API. One of these efforts in peer-to-peer overlay is [10], in which all the common functionality of the underlying libraries is joined as Tier-0 capabilities and offered under a unified API. Part of the community in cloud computing also proposed a common API in [17]. There have been also efforts in the unified runtime systems with the aim to unify heterogeneous multi-core architectures [6] and task scheduling [7].

No unified API has been available, however, for the diverse LWT libraries that exist today.

3 Background

In this section we provide an overview of the most widely adopted stand-alone LWT solutions for HPC.

Qthreads presents a PM with three hierarchical levels composed of shepherds, workers, and work units. In Qthreads, a large number of user-level threads may access any word in memory. Associated full/empty bits are used for synchronizing between ULTs as well as leveraging *mutex* mechanisms. As a drawback, allowing all threads to access any word in memory requires hidden synchronization, which may severely impair performance.

MassiveThreads exposes a recursive-oriented PM. It follows a work-first policy by default, which implies that upon creation, a ULT is immediately executed, pushing the ULT in execution into a *ready queue*. This policy may be configured at library compile time. MassiveThreads exploits the concept of worker as a hardware resource (generally a core), which is created at initialization time. This library does not allow creating ULTs in other threads' queues. Instead, it relies on a work-stealing mechanism.

Argobots is a flexible, mechanism-oriented LWT library. It supports two types of work units: ULTs and tasklets. While the former are the base for all the aforenamed libraries, the latter provides a lighter stackless work unit. Argobots provides the programmer with absolute control of all library resources. Programmers may dynamically create as many execution streams (abstraction of hardware resources) as desired during runtime instead of at initialization. Users can also decide the number of required work unit pools. Although there are default schedulers for each pool, programmers may create their own instances.

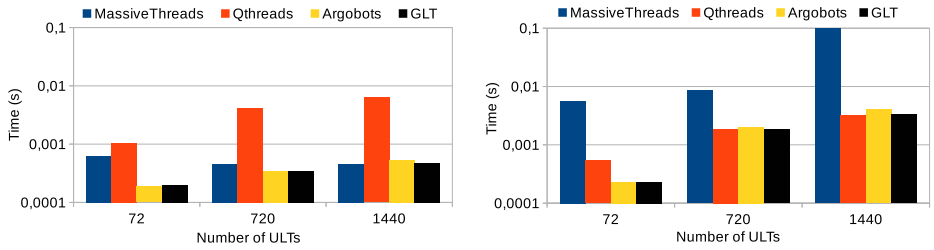
4 Benefits of a Unified LWT API

A unified threading API implemented on top of several underlying libraries avoids having to modify the application code in order to execute it on top of different threading solutions. Different hardware platforms may leverage distinct native LWT libraries for technical or strategic reasons. If more than one is available, users may want to select the library delivering the best performance for their particular case.

To support this assertation experimentally, we have designed two simple microbenchmarks that create fine-grained ULTs. These microbenchmarks are merely created in order to demonstrate how a programmer could benefit from the common API, selecting the desired underlying solution and achieving the best performance possible without modifying the application code.

In the first microbenchmark, each *thread* creates and executes a range of ULTs. In the second microbenchmark, a single *thread* creates all the ULTs, which are executed by all the *threads*. These microbenchmarks have been implemented on top of each native LWT library (Argobots, Qthreads, and MassiveThreads), as well as using the GLT API. Each test has been executed using 72 *threads* with 72, 720, and 1,440 ULTs. The results are the average of 1,000 executions on a 36-core (72-hardware thread) machine equipped with two 18-core Intel Xeon E5-2699 v3 (2.30 GHz) CPUs and 128 GB of RAM. The LWT libraries are Argobots 03-2016, Qthreads version 1.10, and MassiveThreads version 0.95.

Figure 1 shows the microbenchmarks' performance results. Although the GLT implementations are executed on top of the three libraries, only that offering the highest performance is shown. In Fig. 1a this corresponds to GLT over Argobots for 72 and 720 ULTs and to GLT over MassiveThreads for the largest size. In Fig. 1b, on the other hand, GLT over Argobots is the best option for the smallest dataset size, while GLT over the Qthreads library offers the highest performance for the other two problem sizes.



(a) All threads create ULTs.

(b) A single thread creates all the ULTs.

Fig. 1: Performance of the underlying LWT libraries and the best GLT implementation choice when a set of ULTs are created and executed.

These experiments demonstrate the benefits of using a unified LWT API on top of different underlying native implementations. Within the same platform, different LWT libraries may yield distinct performance for different applications. Even the same application may benefit from different LWT implementations depending on the dataset sizes. Therefore, a unified LWT API such as GLT enables users to select the most appropriate underlying native LWT implementation while avoiding the additional work of implementing the same application using several LWT APIs. Determining the best underlying implementation for a particular case is left out of the scope of this paper.

5 GLT Programming Model

As introduced in Sect. 3, each LWT library offers its own PM. Therefore, choosing a correct default PM for GLT is critical.

Figure 2 depicts the set of elements that compose the GLT PM. A *GLT_thread* is composed of the operating system (OS) thread, a queue of ULTs/tasklets, and a scheduler that sets the order of the execution of these work units. The different functionality exposed by their PMs is explained in this section.

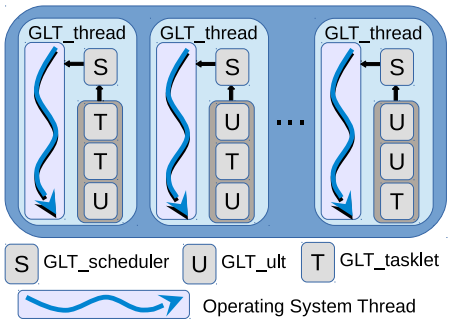


Fig. 2: GLT PM elements abstraction.

A GLT thread executes ULTs in an OS thread. GLT threads are conceptually equivalent to *shepherds* in Qthreads, *execution streams* in Argobots, and *workers* in MassiveThreads. ULTs are conceptually equivalent to *qthreads* in Qthreads and to *threads* in Argobots and MassiveThreads.

GLT sets the environment during the initialization function. By default, one thread is created per CPU core. This number, however, can be defined by the user by means of an environment variable. Each thread is bound to a specific CPU core in the system.

Furthermore, nothing prevents users from changing the default initial resources for the underlying LWT library (e.g., number of pools in Argobots or number of workers per thread in Qthreads) by means of its own environment

variables, which is honored by the GLT implementation. Affinity is always enabled mapping one GLT_thread to each CPU system. No other bindings are allowed due to the GLT PM.

While all our reference libraries provide ULTs, Argobots additionally supports tasklets. Tasklets are lighter than ULTs, but they cannot migrate or yield because a tasklet does not own a stack. These work units are suitable for computation codes that do not include blocking calls. All codes that can be executed by a tasklet can also be executed by a ULT. If GLT is used on top of a library with no native support for tasklets, ULTs are transparently used underneath instead, yielding the expected functionality but no performance benefits.

GLT scheduling relies on the underlying library. This may be specified during the configuration step prior to building those libraries or, as in the case of Argobots, can be changed at execution time.

6 GLT Design and Implementation Details

This section discusses the GLT design choices and describes several implementation details.

6.1 API

GLT objects start with the upper-case prefix "`GLT_`". Table 1 shows the equivalences between the main GLT object types and those of the reference libraries.

Table 1: GLT object equivalences (prefix shown next to each library name).

GLT (GLT_)	Argobots (ABT_)	Qthreads	MassiveThreads (myth_)
ult	thread	aligned_t	thread_t
tasklet	task	aligned_t	thread_t
thread	xstream	qthread_shepherd_id_t	thread_t
mutex	mutex	aligned_t	mutex_t
barrier	barrier	qt_barrier_t	barrier_t
cond	cond	aligned_t	cond_t

GLT functions are organized into modules depending on their functionality. Many GLT functions are simple wrappers to those in the underlying LWT libraries, hence yielding low performance overhead. Some other GLT functions require more elaborate implementations because no direct mapping to the underlying library functionality exists.

GLT is divided into modules that enclose the main necessary semantics. The functionality supported by a complete unified LWT PM is distributed into the following 7 API modules:

– **Setup.** This module initializes and finalizes the library.
– **Work Unit.** It is composed of 18 functions that are used for work unit management. It supports two types of work units: ULTs and tasklets. In case the underlying library does not support tasklets, ULTs are leveraged to deliver analogous functionality.
– **Mutex.** This module includes 5 basic functions to create, destroy, lock, unlock, and try to lock mutexes. Qthreads supports only locking and unlocking natively because of the full/empty-bit mechanism; the remaining functions have been implemented on top of these semantics.
– **Barrier.** Three functions are provided for barrier management.
– **Condition.** Five condition management functions are supported natively by Argobots and MassiveThreads and developed for Qthreads.
– **Util.** It consists of 6 functions to measure elapsed times or to obtain a timestamp and 2 functions that return the number of threads and the rank of the current thread.
– **Key.** This module hosts 4 work-unit data management functions. Natively supported by Argobots and MassiveThreads and implemented for Qthreads.

Although some LWT libraries offer a more complete set of functions, we have included only those that are relevant for the PM we propose. However, we plan to study the addition of the extra functionality if any PM benefits from them.

6.2 Implementations

Our GLT implementation can be used in two ways. On the one hand, a set of *dynamic* libraries compiled on top of the different reference libraries may be generated. This eases the switch among the underlying LWT implementations by linking the application to a different library at load time. On the other hand, we have devised our GLT implementation as a *header-only* library. This second approach offers higher performance than the former because all the functions are labeled as `static inline`. Most compilers will honor these modifiers and prevent the additional function call. The performance result in most cases is analogous to that obtained if the user employs the original library directly, yielding no performance impact for those functions with a direct mapping to the underlying library.

6.3 Semantic Mapping

GLT is largely composed of wrappers to the underlying LWT library functions. The mapping between the most important functions of the GLT API and the reference libraries is shown in Table 2.

The lack of tasklet support in Qthreads and MassiveThreads is compensated with the use of the ULT functions. Moreover, since MassiveThreads does not allow creating ULTs in other workers' ready queue, when a `glt_tasklet/ult_creation_to` is called, the library just creates a ULT in the current worker's

Table 2: Mapping between some GLT functions and their equivalent in the underlying libraries (prefix shown next to each library name).

GLT (glt_)	Argobots (ABT_)	Qthreads (qthread_)	MassiveThreads (myth_)
tasklet_creation	task_create	fork	create
ult_creation	thread_create	fork	create
ult_creation_to	thread_create	fork_to	create
yield	thread_yield	yield	yield
ult_join	thread_free	readFF	join

queue. Despite the fact that the different implementation approaches over different underlying native LWT libraries may have performance implications, these all conform to the exposed GLT semantics (offering the same functionality to GLT users) while transparently leveraging the most efficient mechanism underneath.

7 Performance Evaluation

We next compare the performance of our test cases implemented directly on top of the low-level libraries with the codes that use the GLT API. The results correspond to the average of 1,000 executions. The software and hardware configuration employed was introduced in Sect. 4.

7.1 Microbenchmarks

We leverage the Callgrind profiling tool [18] to measure the overhead in terms of instructions per call of the most frequently used functions of the GLT code for our three reference LWT libraries. These functions are initialization (`Init`), work unit allocation (`Malloc`), work unit creation (`Creation`), yield (`Yield`), join (`Join`), and number of threads query (`Num_thr`).

Figure 3 shows our results for the Qthreads, MassiveThreads, and Argobots GLT implementations, comparing the results with the native approaches. The plots expose a common pattern: `Init`, `Malloc`, and `Creation` show a small increment in the number of instructions in both GLT variants (*dynamic* vs *static*); but `Yield`, `Join`, and `num_threads` experience this increment only in the stand-alone version of GLT. These results reflect that the second group of functions contains pure wrappers to the original functions and that the additional function call overhead is added only in case of leveraging a separate GLT library. The library initialization function adds a relatively high number of instructions because of the GLT environment set up. Nevertheless, this is a one-time overhead introducing merely 10–15% additional instructions compared with the native LWT solutions. The `Malloc` overhead (up to 4 instructions per call) is caused by the type casting of the value returned by the allocation function to the appropriate work unit pointer. The instructions added in `Creation` are due to the function

pointer casting and the return of the work unit handler. These results confirm that the use of the GLT library as a high-level LWT API introduces fairly low overhead.

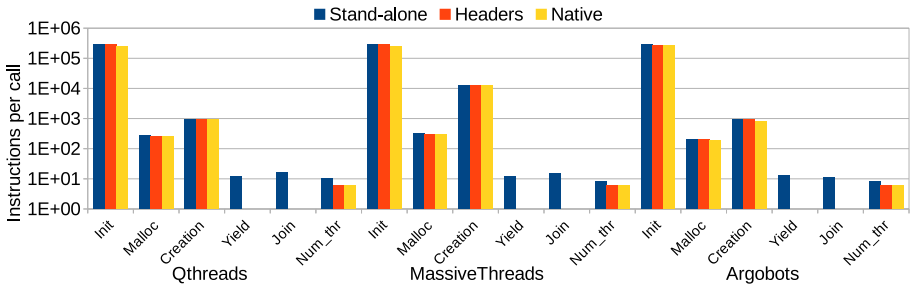


Fig. 3: Overhead (%) GLT approaches compared with overhead (%) native libraries.

7.2 N-Queens

We evaluated the overhead of the GLT API using a translation from an OpenMP version of N-Queens [11]. The number of lines of code needed in the translation are 185 for Argobots code compared with 158 for Qthreads, MassiveThreads, and GLT. Our unified API does not add more lines to the code; indeed, it even reduces the number compared with Argobots. The reason is the automatic environment setup described in Sect. 6.

In the base OpenMP implementation, a single thread creates the first set of tasks (to place a queen in a cell) and executes a `taskwait`. Each task creates more tasks and waits for their termination. Our implementation of this algorithm using LWTs follows the same philosophy. The main thread creates the first work units, and each of these is placed into other threads' queue until each thread has at least one work unit to be executed. Once that is completed, each thread creates its own work. The threads wait for the finalization using the *join* function.

Table 3 summarizes the average overhead of several thread configurations (from 1 to 72 threads), for three problem sizes—10, 11, and 12 queens—and the reference LWT libraries. While the average overhead for the stand-alone version varies from 0.28% to 0.56%, for the header-only GLT deployment this overhead is less than 0.1%.

These results showcase the low overhead introduced by the use of the GLT API. The results also show a constant behavior that indicates that the overhead is not caused by the size problem. The largest cost with respect to the native implementations is under 0.6%.

Table 3: Average overhead (%) executing the N-Queens application using *headers* (H) and *stand-alone* (S) GLT implementations over the three libraries.

GLT Underlying Library (Mode)	Number of Queens		
	10	11	12
Argobots (H)	0.01	0.06	0.04
Argobots (S)	0.28	0.36	0.32
MassiveThreads (H)	0.02	0.01	0.00
MassiveThreads (S)	0.48	0.33	0.49
Qthreads (H)	0.08	0.08	0.09
Qthreads (S)	0.43	0.51	0.56

7.3 UTS Benchmark

UTS Benchmark is a parallel code that measures the performance attained when executing an exhaustive search on an unbalanced tree. The tree is built at execution time by using a divisible random number generator that splits the structure, making possible the parallel processing while still generating a deterministic tree. We translated the original code written in Pthreads to our GLT API using 71 code lines for the Argobots implementation and 38 for MassiveThreads, Qthreads, and GLT.

In the original Pthreads implementation, the main thread initializes the tree and places the first (tree) node into its own queue. Then all threads execute the same function. First, the next node in the queue is executed, and this node creates more nodes that are pushed into the local queue. If its local queue is empty, a thread tries to steal a certain number of nodes from other queues.

In our implementation, a work unit is created for each thread, and work-stealing is performed as in the original code. Accessing other threads' queues requires synchronization among threads and is done via *GLT_mutex*.

In this scenario, GLT can leverage the lighter tasklet work unit because the code does not include any blocking or system call. As discussed in Sect. 6, GLT implementations over MassiveThreads and Qthreads employ ULTs instead of tasklets. For reference, we also include the results for native Argobots based on ULTs.

We calculated the average overhead for all the executions of different problem sizes in order to obtain a global vision of the overhead introduced by the GLT API. Table 4 shows the average overhead when executing the UTS benchmark with problems T1, T1L, T1XL, and T1XXL (of 4 million, 102 million, 1.6 billion, and 4.2 billion nodes, respectively), on top of the three underlying libraries, modifying the number of threads from 1 to 72. As in the N-Queens case, the difference using the stand-alone (S) and header-only (H) GLT versions is perceivable, being under 0.6% for the former and just slightly above 0.1% for the latter. The results also show a trend that does not correspond with the problem size, so it indicates that the overhead is not caused by the size problem.

Table 4: GLT average overhead executing the UTS benchmark using headers (H) and stand-alone (S) GLT implementations over the three underlying libraries.

GLT Underlying Library (Mode)	Problem Size			
	T1	T1L	T1XL	T1XXL
Argobots Task (H)	0.06	0.00	0.01	0.00
Argobots Task (S)	0.08	0.36	0.39	0.28
Argobots ULT (H)	0.03	0.01	0.01	0.00
Argobots ULT (S)	0.24	0.55	0.22	0.53
MassiveThreads (H)	0.11	0.00	0.08	0.05
MassiveThreads (S)	0.45	0.50	0.45	0.18
Qthreads (H)	0.00	0.01	0.02	0.06
Qthreads (S)	0.30	0.55	0.58	0.29

8 Conclusions

In this work we have introduced the GLT API [1]. This library proposes a unified API for LWT solutions that is the first attempt to standardize those PMs. Moreover, we have implemented GLT on top of the major general-purpose LWT solutions for HPC: Argobots, MassiveThreads, and Qthreads.

In addition, we have discussed the GLT PM and decomposed the API's modules. Furthermore, we have presented an example of the semantic mapping between the GLT API with the LWT solutions. Using two microbenchmarks we have also justified the need for a unified LWT API from the point of view of portability.

Our performance evaluation, based on stand-alone and header-only implementations of the GLT API, demonstrates the low performance overhead of this approach. We have demonstrated this overhead with a set of microbenchmarks that measure the instructions per call added with GLT. Moreover, we have assessed the overhead by comparing the execution time of two applications where, the stand-alone implementation produced an average overhead under 0.6%, while the header-only version showed an average overhead below 0.1%.

In conclusion, we have demonstrated the portability benefits that a unified API for LWT libraries can offer to programmers translating their applications from OpenMP and Pthreads to GLT API. As part of future work, we plan to implement several high-level PMs on top of the GLT API, such as OpenMP or OmpSs. Moreover, we plan to augment the API with additional functionality that some PMs/applications can benefit from.

Acknowledgments. Researchers from the Universitat Jaume I de Castelló were supported by project TIN2014-53495-R of the MINECO, the Generalitat Valenciana fellowship programme Vali+d 2015, and FEDER. Antonio J. Peña is cofinancied by the Spanish Ministry of Economy and Competitiveness under Juan de la Cierva fellowship number IJCI-2015-23266. This work was partially

supported by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357.

References

1. Generic Lightweight Thread. `github.com/adcastel/GLT`
2. Programming with Solaris Threads. `docs.oracle.com/cd/E19455-01/806-5257/`
3. Pthreads API. `https://computing.llnl.gov/tutorials/pthreads/`
4. MPICH, High-Performance Portable MPI. `http://www.mpich.org/` (2016)
5. TOP500 Supercomputer Sites. `www.top500.org/` (June 2016)
6. Augonnet, C., Namyst, R.: A Unified Runtime System for Heterogeneous Multicore Architectures, pp. 174–183. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
7. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011)
8. BSC: Nanos++, `pm.bsc.es/projects/nanox/`
9. Castelló, A., Peña, A.J., Seo, S., Mayo, R., Balaji, P., Quintana-Ortí, E.S.: A review of lightweight thread approaches for high performance computing. In: IEEE Int. Conf. on Cluster Computing. Taiwan (September 2016)
10. Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., Stoica, I.: Towards a common API for structured peer-to-peer overlays. In: The Second Int. Workshop on Peer-to-Peer Systems. Springer Berlin Heidelberg (2003)
11. Duran González, A., Teruel, X., Ferrer, R., Martorell Bofill, X., Ayguadé Parra, E.: Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: 38th International Conference on Parallel Processing. pp. 124–131 (2009)
12. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 97–104. Springer (2004)
13. Kalé, L.V., Bhandarkar, M.A., Jagathesan, N., Krishnan, S., Yelon, J.: Converse: An interoperable framework for parallel programming. In: Proceedings of the 10th International Parallel Processing Symposium (IPPS). pp. 212–217 (April 1996)
14. Microsoft MSDN Library: Fibers
15. Nakashima, J., Taura, K.: MassiveThreads: A thread library for high productivity languages. In: Concurrent Objects and Beyond, Lecture Notes in Computer Science, vol. 8665, pp. 222–238. Springer Berlin Heidelberg (2014)
16. Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P., Castelló, A., Genet, D., Herault, T., Jindal, P., Kalé, L.V., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., Beckman, P.: Argobots: A lightweight threading/tasking framework (2017), https://collab.cels.anl.gov/display/ARGOBOTS/
17. Silva, L.A.B., Costa, C., Oliveira, J.L.: A common API for delivering services over multi-vendor cloud resources. Journal of Systems and Software 86(9), 2309–2317 (2013)
18. Valgrind Developers: Callgrind: A call-graph generating cache and branch prediction profiler (2010)

19. Van Zee, F.G., van de Geijn, R.A.: BLIS: A framework for rapidly instantiating BLAS functionality. ACM Trans. Math. Softw. 41(3) (2015)
20. Wheeler, K.B., Murphy, R.C., Thain, D.: Qthreads: An API for programming with millions of lightweight threads. In: Proceedings of the 2008 Workshop on Multithreaded Architectures and Applications (MTAAP) (April 2008)