Towards Runtime Support for Domain-Specific Languages of Adaptive and Irregular Applications

Polykarpos Thomadakis and Nikos Chrisochoides
{pthomadakis,nikos}@cs.odu.edu

Old Dominion University, Norfolk VA 23508, USA

Abstract. We present performance and ease-of-use improvements to a runtime for Domain Specific Languages of irregular applications. Support for message-driven global address space is integrated with lightweight threads which in combination with fine-grained concurrency improves the system's performance and load balancing in shared and distributed memory. We observe up to 100% difference in performance behavior for different lightweight thread creation strategies. Evaluations on a 1960-core distributed memory machine show that the integration of fine-grained concurrency with the runtime achieves performance improvements of 12% on a seismic wave simulation benchmark, as opposed to 50% degradation with OpenMP. Studies on workload decomposition on the same benchmark showed that over-decomposition on both data and task level produces the best results.

Keywords: Active Messages, Distributed Computing, Tasking, Adaptive and irregular applications, Runtime support software.

1 Introduction

Developing efficient applications for modern, large scale, heterogeneous computing platforms require a lot of effort and expertise in both the application and the systems domain. This is even more relevant for unstructured and irregular applications whose workflow is not statically predictable and heavily depends on the input. Our solution is the introduction of a high-level Domain Specific Language (DSL) that hides the idiosyncrasies of the hardware and the effort required for maintaining correctness, to transparently scale applications. In this work, we focus on the runtime framework, namely the Parallel Runtime Environment for Multicore Applications (PREMA) [1, 2] that serves as its backend.

An effective runtime should address the following fundamental issues in parallel computing: a) global namespace, b) scheduling and load balancing, c) latency hiding, d) fault resilience, e) heterogeneity and performance portability. Currently, PREMA addresses the former three while work is in progress for the latter two. In the process of designing and implementing high-level DSL constructs on top of PREMA, we realized some of its limitations, also found in other

similar systems, that inhibit its performance and ease-of-use. These limitations include: 1) remote method invocations (Active Messages) or tasks have to run to completion to avoid delaying the progress engine, 2) inability to preempt task execution, 3) lack of fine-grained parallelism inside remote method invocations. We overcome these limitations by integrating PREMA with Argobots [3].

Parallel Runtime Environment for Multicore Applications (PREMA) supports applications targeting large-scale computing platforms. Its goal is to alleviate users from the burden of dealing with work scheduling and load balancing on both shared and distributed memory. To achieve this, PREMA introduces a mobile object-driven (MOD) programming model where interactions are expressed as remote method invocations (handlers in PREMA) between mobile objects rather than processes or threads. A mobile object is a location-independent container that encapsulates semi-isolated, coarse-grained application data which can be located anywhere in the distributed memory and can be implicitly moved by the runtime. Handlers can be invoked on mobile objects uniformly, regardless of whether their data are local or remote.

By utilizing the MOD programming model and associating remote handlers with access privileges, an application is able to transparently run on multi-core distributed platforms without explicitly handling concurrency. PREMA is able to extract shared-memory parallelism by running non-conflicting handlers concurrently and allowing threads to shared their workload. On the distributed memory level, it can migrate mobile objects between different computing nodes in order to provide distributed-memory load balancing. To increase flexibility, the framework exposes a simple and isolated module that allows easy experimentation/development of new 2-level load balancing/scheduling policies without affecting the application code.

Argobots is a low-level, lightweight, threading, and tasking framework developed with exascale computing platforms in mind. Argobots' execution model consists of two levels of parallelism: Execution Streams (ESs) that map to a hardware thread and guarantee progress, and Work Units (WUs) which represent execution units running within a ES. A WU can either be a User-level Thread (ULT), which has its own stack and can explicitly yield, or a Tasklet, which shares the stack of an ES and runs to completion. Apart from operations to create, join, and yield a ULT, Argobots provide a set of synchronization tools (e.g., mutexes) that leverage the ULTs' ability to yield. Libraries such as OpenMP, Intel TBB[1], and others [4–6] also deliver fine-grained and efficient task scheduling. However, they provide little to no control over task scheduling, hide thread pools, and do not allow thread yielding explicitly. Argobots provide all these and also incorporate hooks for integration with MPI and power management systems which renders it the best choice for our runtime.

Contributions This paper presents an effort to address challenges related to message-driven runtime frameworks, like PREMA, by using lightweight threads

[1] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html

tightly integrated with message-passing. We use Argobots to demonstrate a number of optimizations, including threads capable of preemption. In [7], we demonstrate that Argobots perform on par or better than OpenMP and TBB as a tasking framework in the context of Adaptive Speculative Parallel Mesh Generation. We extend this work to provide distributed intra-handler parallelism, allowing applications to further decompose handlers into shared tasks (tasklets) that avoid resource over-subscription and are able to utilize PREMA's underlying threads. We experiment with different approaches for handler/task creation, and utilize lightweight threads that allow blocking in handler invocations while avoiding possible cases of live-locks and other pertinent latencies. Finally, we evaluate the performance of our fine-grained tasking module on top of PREMA on a seismic wave simulation benchmark. We observe significant performance improvements when exploiting domain and task over-decomposition.

2 Related Work

Distributed memory runtime systems have been used since the beginning of the field of High-Performance Computing [8]. Systems like Split-C [9] and UPC [10] introduced the partitioned global address space (PGAS) environment for parallel computing as an extension to the C language, using globally accessible arrays distributed among the computing nodes. Hiding message-passing into global array accesses makes it feasible to achieve functional programs by sharing a common virtual address space; however, it is difficult for developers to optimize remote accesses since they are implicit. In addition, data allocation is static and cannot change based on dynamic load. Titanium [11] made inter-node communication explicit; however, data migrations are still not supported.

Chapel [12] extends the PGAS languages model with an asynchronous approach and the abstraction of locales. Locales can be either an abstracted or real machine component where data or computations can reside; work and data are then assigned to them explicitly. Even though Chapel supports shared memory tasking, distributed load balancing has to be explicitly implemented by the application developer. HPX [13] is another task-based runtime using the asynchronous PGAS approach. HPX does not support distributed load balancing, although it does so in the shared memory. Thus, these systems are not suitable for developing irregular applications that require dynamic data redistributions.

3 Integration of PREMA and Argobots

In this section, we present the integration PREMA's software layers, namely the Data Movement and Control Substrate (DMCS), the Mobile Object Layer (MOL), and the Implicit Load Balancing (ILB) with Argobots.

DMCS incorporates MPI to utilize multiple computing nodes in a high-performance computing platorm, as well as PThreads to take advantage of the hardware cores of each node. DMCS provides functionality similar to Active Messages, i.e., each message sent is associated with a function call to be invoked

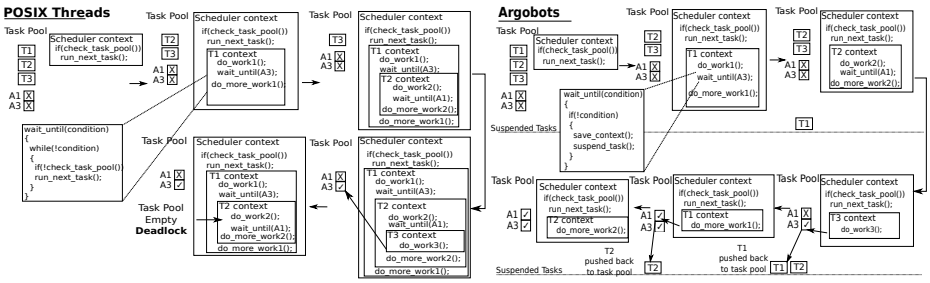


Fig. 1. An example of a blocking handler causing deadlock (left), and how using Argobots solves this issue (right).

on the receiver side once the message is received. This layer is divided into three components: the application, the communication, and the handler-execution component. The application component runs on the default, implicitly-created Argobots ES. The handler-execution component consists of multiple ESs, usually one less than the number of cores residing on a computing node, responsible of executing remote handlers. The application and handler-execution components run customized Argobots schedulers, each assigned with a primary work pool while accessing each others' pools for work-stealing. The communication component encapsulates all the message-passing related operations and is either handled by a dedicated stream or the handler-executing and application components when the rest of their work has been completed. When a remote method invocation request is received in the communication component, a new ULT is created that is pushed to a randomly selected pool of the handler execution component or to the application component's work pool. In section 5.1, we evaluate different approaches for handler/task creation using Argobots.

By encapsulating remote handler invocations in ULTs their execution can be interrupted at any point allowing others handlers to execute on the same hardware thread. Using the yielding functionality, an application can suspend a running ULT that has to wait for some resource to mitigate latencies. In contrast, in the PThreads implementation, using busy waiting inside a handler for reasons other than for sending a message was discouraged as it could cause starvation or even deadlocks. The issue arises by the implementation of the *wait_until([condition])* operation, which blocks the running thread until the given condition evaluates to true. To avoid wasting cycles while waiting, PREMA tries to find another task to run by popping the next task available in the work-pools; however, when used from inside a handler, it can lead to live-lock cases.

Let us assume a scenario of three tasks, namely T1, T2, and T3 (see Figure 1 left) where T1 needs to wait for some acknowledgment A3 from T3, T2 waits for acknowledgment A1 from T1, and T3 does not wait for any acknowledgments. PREMA starts running T1 until it blocks waiting for acknowledgment A3, then switches to T2 until it blocks waiting for acknowledgment A1, and finally switches to T3, which acknowledges A3. Even though A3 has been ac-

knowledged, the control will never return to T1 to unblock it. Once T3 finishes its execution, the control returns to T2's *wait_until()* operation, which will keep checking the task pool but will never run T1 since T1 has already been popped and is running T2 from its *wait_until()* operation. The Argobots implementation avoids such a scenario by using separate stacks for each task, saving their states before switching control of the execution stream, and resubmitting them to the task pool when unblocked. This also allows blocked tasks to be stolen in case the currently running thread starts a long-running process.

The ILB layer is implemented as an Argobots' stackable scheduler that is pushed to the dedicated pool of each available execution stream to change the DMCS scheduling policy. It inherits the pools created by DMCS to continue executing remote handlers of the lower layers while also handling pools dedicated to the ILB. Handlers need to be issued through the ILB messaging operation for their loads to be monitored. Such requests are routed to the current location of the target mobile by the MOL. The ILB is notified about the new handler, the handler's load is calculated, and it is pushed to the list of pending work of the target mobile object. The scheduler maintains the workload of all local mobile objects and, thus, the workload of the each node. To schedule new work it picks a pending handler from the next available mobile object and creates a ULT. ULTs created at this level are then pushed to the fine-grained tasking module, presented in section 4. If no pending handlers are available, ILB starts a phase of distributed load balancing that might result in the migration of mobile objects along with their workload. The distributed memory load-balancing scheduler is also implemented as a ULT, allowing load balancing policies to use blocking operations without affecting performance.

4 Fine-grained Recursive Task Parallelism

The need for finer-grained parallelism inside a handler arises from the large workload disparity witnessed among handlers of irregular and adaptive applications. Even though workload is diffused through decomposing application data into multiple mobile objects, the difference among handlers targeting different mobile objects can still be large. To cover this gap, we decided to develop a stand-alone tasking parallelism module on top of Argobots that can run as part of a remote handler or independently, in order to utilize multiple hardware threads in the context of a single handler execution. When used as an integrated part of PREMA, this module can utilize the existing execution streams, avoiding the creation of new threads and the possible over-subscription overheads. To distinguish between ULTs used in other parts of PREMA and tasks created by this module, the latter will be called tasklets for the rest of the paper.

The goal of the tasking framework's scheduling policy is to maximize parallelism and minimize memory use by using a hybrid of depth-first and breadth-first execution policy and recursive creation of work units. Each processing element (e.g., ES) is associated with a work-pool of tasklets. Each time a new tasklet is created, it is pushed to the bottom of the PE's pool. To execute a new

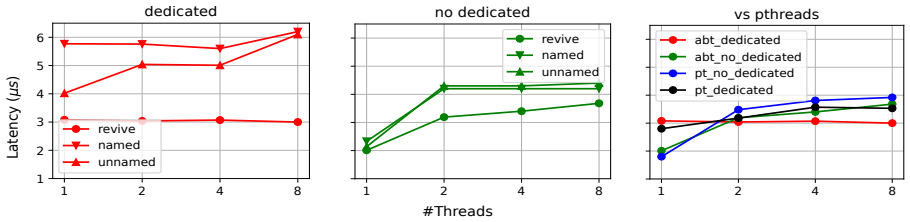


Fig. 2. Latency observed on ping pong benchmark for different task creation approaches using dedicated streams for communication (a) or not (b) and comparison of the best approaches with the PThreads implementation (c).

tasklet, the PE simply pops one from the bottom of its pool; if empty, it tries to steal from the top of another PE's pool. Provided that the tasklets creation is performed recursively, this scheduling algorithm minimizes memory consumption and prioritizes tasklets that are hot in the cache when the tasklet pool is not empty. When stealing is required, the policy maximizes the amount of stolen work by picking tasklets that were created early in the recursion steps.

To implement this scheduling algorithm, the Argobots abstraction of custom pools was used to encapsulate a lock-free implementation of a circular double-ended queue (deque)[14]. The abstract pools API only provides push and pop operations[2] to manipulate their contents, thus, stealing is implemented as part of the pop operation, and each abstract pool is implemented as an array of pointers to all available deques, instead of associating it with a single deque. When an ES is ready to pick a new task, it calls the pop function, which checks its deque and, if empty, randomly steals a tasklet from another deque.

5 Performance Evaluation

In this section, we measure the performance of PREMA using synthetic microbenchmarks as well as real world applications. The computing platform used is a 200-node cluster which utilizes Intel(R) Xeon(R) Gold 6148 @ 2.4 GHz CPUs of 40 cores each in two sockets (4 NUMA nodes). We use OpenMPI version 3.1.4, Argobots version 1.0 and gcc version 7.5 for all benchmarks.

5.1 Handler Task Creation

In this subsection, three approaches to creating handler tasks are examined. Performance is measured as the latency in handler creation, with and without a dedicated communication stream, in a ping pong benchmark. Two nodes exchange 20000 64B-sized messages in total, where the sender sends a message and then waits for an acknowledgment. The three approaches are described below:

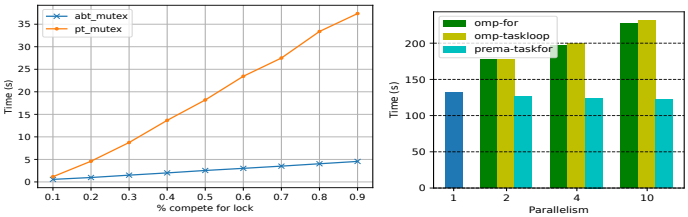
[2] They also provide a remove operation but is not needed in our case

- Unnamed: The Argobots runtime is responsible for monitoring and releasing the memory of ULTs when they complete.
- Named: PREMA checks ULTs for completion and frees their resources explicitly. An array of handles is maintained per handler executing ES.
- Revive: A variation of the second approach where the completed ULTs in the preallocated arrays are reused through the *ABT_revive()* function.

Figure 2a shows the latency observed when using a dedicated communication stream. The "revive" approach performs best, maintaining a latency of $3\mu s$. In contrast, "named" and "unnamed" achieve lower performance (of 6 and 4 μs, respectively) and increasing overheads as the number of streams increases. The performance degradation comes from the fact that new ULTs require a new stack memory to be allocated, increasing the critical path of ULT creation. In contrast, "revive" reuses the memory of previously completed ULTs. When no dedicated stream is used (Figure 2b) the three approaches have a lower effect, especially on a single stream because Argobots use memory pools for the stacks of completed ULTs per ES. When a ES executes a ULT, it stores the stack memory in its memory pool for later use to avoid allocations. However, when one ES creates most ULTs (producer) which other ESs execute (consumers), the producer's pool is depleted without being reused since consumers keep the stack memory in their own pools, forcing the producer to steal/allocate memory. This is observed in "named" and "unnamed" when two or more streams are used but in a less significant level than in Figure 2a because all streams have the same chance of producing/consuming ULTs and, thus, refilling their pools. Figure 2c compares the revive approach with the PThreads implementation. For a single thread, using no dedicated thread exhibits the least overhead and PThreads achieves the best performance. When more threads are available, using a dedicated stream shows lower latency for both the Argobots and PThreads implementation, with Argobots achieving 15% better performance. A slightly lower performance is observed without a dedicated stream, with Argobots exceeding PThreads' performance by up to 10%.

5.2 Blocking Operations in Handler Execution

An important feature stemming from the integrating with Argobots is the ability to yield handler execution either explicitly by calling the respective function or implicitly when a blocking call is detected. The potential benefit provided is presented through a synthetic benchmark. An allocation of ten cores is used along with ten mobile objects, each using an exclusive mutex to provide access to its data. For each mobile object, 100 handler invocations are issued where a specific percentage of them needs to take control of the mutex before executing. We set the time that the mutex is held to 50ms at a time and experiment with standard PThread mutexes and ULT-aware mutexes that can suspend handlers when locked. Figure 3a shows an evaluation of the two implementations with different percentages of handlers acquiring the mutexes, ranging from 0.1 to 0.9 (10 - 90 handlers/mobile object). One can observe the tremendous difference



(a) (b)

Fig. 3. (a) Execution time with respect to percentage of tasks competing for the same mutex. (b) Intra-handler parallelism with OpenMP for, taskloop and PREMA tasklets.

observed in the two implementations (up to 1000%). The issue with the PThread implementation is that a handler that tries to access a taken mutex will block the hardware thread it executes on, preventing handlers targeting a different mobile object/mutex from running. In contrast, the Argobots implementation will suspend the running handler ULT, allowing another handler to run on the same thread. In similar cases, the user might know that acquiring a resource exclusively could cause delays and, thus, may explicitly yield competing handlers to mitigate the propagation of such effects.

5.3 Seismic Wave Simulation Benchmark

In this section, we evaluate the new features of PREMA on the SW4lite[3] benchmark. We study its performance on different work unit allocations (mobile objects, PREMA tasklets) for multi-node experiments. In Figure 3b, we exploit inter-/intra-node parallelism using one mobile object per core for 640 cores, and also explore intra-handler parallelism using OpenMP and PREMA tasklets. Using OpenMP causes over-subscription of hardware resources since it is unaware of the threads already running in PREMA, resulting in high overheads. In contrast, tasklets efficiently utilize the existing threads, achieving performance up to 10% better over the base case and 60% over OpenMP. Next, we perform more experiments with work unit allocations to explore how different combinations of domain and task decomposition can affect the performance of an application. We evaluate (over-)decomposion at domain level, task level and a combination of the two. The different work unit allocations allow different levels of freedom for PREMA to take advantage of (e.g., load balancing and latency hiding) since both domains and tasklets in a PREMA instance are shared among its threads. Figure 4 presents studies for different core allocations where the overall running time is normalized by the run time achieved when using one mobile object per hardware core, one tasklet per handler, and one instance of PREMA per 10 hardware cores (base case). The x-axis shows the work-unit allocations as a pair of the number of mobile objects per PREMA instance and number of tasklets per

[3] https://github.com/geodynamics/sw4lite

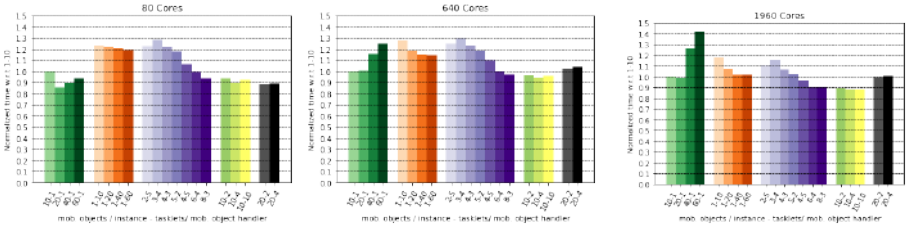


Fig. 4. Normalized performance of the SW4 benchmark for different PE allocations with respect to the performance achieved when mapping one mobile object per PE and one tasklet per handler (10-1).

mobile object handler. As can be seen from the figure, the application benefits from increasing the number of mobile objects per instance (green bars) when the number of cores is low and the work enclosed in each handler is substantial but this benefit degrades as the number of cores increases and the work per handler decreases. Utilizing task decomposition only (orange bars) falls short, but its performance improves as the number of taskles increases and the size of data domains per PREMA instance decreases. Combining the two approaches has better results when the number of mobile objects is close to the number of threads, and task decomposition is performed on top of that (purple bars). The best results (12% improvement) in most cases were achieved when using a combination of 10 mobile objects and 10 tasklets (yellow bars).

6 Conclusion and Future Work

We have presented the integration of PREMA with lightweight threads utilizing Argobots. The product of this effort overcomes the previously exhibited limitations while incorporating a tasking framework that allows for fine-grained intra-handler parallelism. We have experimented with multiple design choices for task creation where we observed up to 100% difference in latency. Moreover, we have shown that the lightweight threads remove constraints of previous implementations while mitigating overheads of synchronization semantics. The tasking framework achieves performance better than OpenMP by up to 60% when used in the context of PREMA. Our experimentation with different combinations of workload decomposition both on the domain and task level showed up to 12% improvements on the SW4lite benchmark. The product of this work allows for an easier-to-use and more efficient runtime backend for our new DSL. In the future we plan to extend this framework to handle needs such as heterogeneity and fault-tolerance arising from the emerging computing platforms.

Acknowledgments

This work is funded in part by the Dominion Fellowship, the Richard T. Cheng Endowment at Old Dominion University and NSF MRI grant no: CNS-1828593.

The authors would like to thank the Argo team and especially Dr. P. Beckman and Dr. S. Iwasaki for their support in configuring Argobots to accommodate PREMA's extreme requirements.

References

1. K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali, "A load balancing framework for adaptive and asynchronous applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, pp. 183–192, February 2004.
2. P. Thomadakis, C. Tsolakis, K. Vogiatzis, A. Kot, and N. Chrisochoides, "Parallel software framework for large-scale parallel mesh generation and adaptation for cfd solvers," in *AIAA Aviation Forum 2018*, (Atlanta, Georgia), June 2018.
3. S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2018.
4. J. Nakashima and K. Taura, *MassiveThreads: A Thread Library for High Productivity Languages*, pp. 222–238. Berlin, Heidelberg: Springer, 2014.
5. K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *IEEE Int. Symposium on Parallel and Distributed Processing*, pp. 1–8, 2008.
6. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.
7. C. Tsolakis, P. Thomadakis, and N. Chrisochoides, "Tasking framework for adaptive speculative parallel mesh generation," *J. Supercomput.*, vol. 78, pp. 1–32, 2022.
8. P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *J. Supercomput.*, vol. 74, p. 1422–1434, apr 2018.
9. A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in Split-C," in *Int. Conf. on Supercomputing*, p. 262–273, ACM, 1993.
10. W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," tech. rep., UC Berkeley, 1999.
11. K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high performance Java dialect," *Conc. - Pract. & Exp.*, vol. 10, pp. 825–836, 1998.
12. B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perf. Comp. Appl.*, vol. 21, pp. 291–312, Aug. 2007.
13. H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Int. Conf. on Partitioned Global Address Space Programming Models*, pp. 1–11, ACM, 2014.
14. D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, p. 21–28, ACM, 2005.