Received: 00 Month 0000 | Revised: 00 Month 0000 | Accepted: 00 Month 0000

DOI: xxx/xxxx

NEW MANUSCRIPT

Efficient Lock-Step Synchronization in Task-Parallel Languages

Akshay Utture[1] | V Krishna Nandivada*[2]

[1]Dept of CSE, IIT Madras, Chennai, India, Email: akshay.utture@gmail.com
[2]Dept of CSE, IIT Madras, Chennai, India, Email: nvk@iitm.ac.in

Correspondence
*V Krishna Nandivada;
Email: nvk@iitm.ac.in

Many modern task-parallel languages allow the programmer to synchronize tasks using high level constructs like barriers, clocks and phasers. While these high level synchronization primitives help the programmer express the program logic in a convenient manner, they also have their associated overheads. In this paper, we identify the sources of some of these overheads for task-parallel languages like X10 that support lock-step synchronization, and propose a mechanism to reduce these overheads.
We first propose three desirable properties that an efficient runtime (for task-parallel languages like X10, HJ, Chapel, and so on) should satisfy, to minimize the overheads during lock-step synchronization. We use these properties to derive a scheme to called uClocks to improve the efficiency of X10 clocks; uClocks consists of an extension to X10 clocks and two related runtime optimizations. We prove that uClocks satisfy the proposed desirable properties. We have implemented uClocks for the X10 language+runtime and show that the resulting system leads to a geometric mean speedup of 5.36× on a 16-core Intel system and 11.39× on a 64-core AMD system, for benchmarks with a significant number of synchronization operations.

KEYWORDS:
Lock-step synchronization, Task-parallel languages, Runtime Optimizations

1 | INTRODUCTION

1.1 | Motivation

The power wall has been effectively bringing multi-core systems and the associated task-parallel programming to the mainstream. Task-parallel languages like X10[1], HJ[2], Chapel[3], and so on, allow the programmer to specify task synchronization, using high level constructs like clocks, phasers and barriers. These languages also provide the necessary runtime libraries to facilitate low level synchronization and thread management.

For example, X10 provides light-weight tasks, (*a.k.a* asynchronous activities); the runtime maps these tasks to threads and schedules them efficiently. In addition, X10 supports fine-grained synchronization using the feature of clocks (similar to that of phasers in HJ, barriers in Chapel), which lets tasks make progress in lock-step synchrony. Yuki et al.[4] show that these constructs help write arguably more natural and readable codes. However, these high level constructs (and their implementations) can lead to some overheads. We explain the same using an example.

Figure 1 shows a snippet of X10 code that computes parallel one-dimensional iterative averaging; a minor variation of code shown by Shirako et al[5]. In X10, an `async` creates an asynchronous task that can run in parallel with the parent task. The `clocked` construct registers the task to the list of clocks (`c`). Here each of the `n` tasks execute a serial loop and synchronize

```
delta = epsilon+1;
iters = 0;
finish{
        c = Clock.make();
        for (j in 1..n ) {
            async clocked (c) {
                while ( delta > epsilon ) {
                    newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
                    diff[j] = Math.abs(newA[j]-oldA[j]);
                    c.advance();
                    atomic {
                        delta += diff[j];
                        ...
                    }
                    if (j==1) {
                        iters++;
                        temp = newA; newA = oldA; oldA = temp;
                    }
                    c.advance();
                }
            }
        }
}
```

FIGURE 1 Motivating example. A snippet of X10 code.

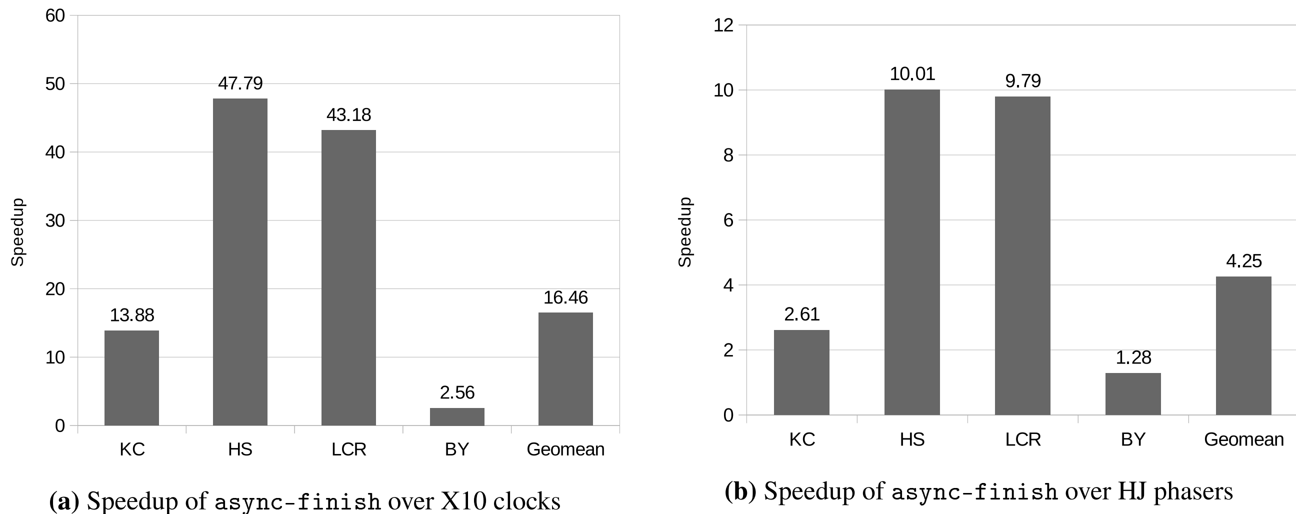
with each other using clocks by calling `c.advance`, which brings in the notion of phases. Here, each of the `n` tasks execute the statements before the `advance` (compute `diff[j]`), and wait for each other (for phase completion), before proceeding to the next phase. The `atomic` construct enforces mutual exclusion among the tasks of the program. Thus, the variable `delta` is updated in a mutually-exclusive manner by each task. See Section 2 for detailed X10 syntax.

Executing codes like the one shown in Figure 1 exposes a number of interesting insights and their corresponding issues related to performance. We focus on three of them.

1. Lock-contention: The implementations of `advance` and `atomic` need to execute some code in a critical section and that part can be implemented in the runtime by using a common mutual-exclusion library. For example, in XRX (X10 Runtime implemented in X10)[6] the mutual exclusion part in `advance` is achieved by executing the critical section code inside an `atomic` block. Though the design looks harmless at the first glance, this leads to an unintended contention among the tasks executing `atomic` (updating `delta`, in Figure 1) and `advance` that may otherwise run in parallel.

2. Excessive thread context-switching: The X10 runtime initially creates X10_NTHREADS number of threads to execute the `n` tasks; each thread runs a task to completion (without switching to any other task in between), and then picks up the next task to execute. However, when a thread executing a task reaches an `advance`, it blocks itself (no switching), and starts a new thread to execute the remaining tasks. Finally, when the last task executes the `advance`, all the blocked threads (total `n`, in Figure 1) are woken up and may start executing in parallel. This can lead to large context switching overheads because `n` could be much larger than the number of available hardware cores.

3. Eager waking: In the current X10 runtime, every time a thread executes `advance` it wakes up the blocked threads ($O(\mathtt{n})$, in Figure 1), who in turn check if the phase is complete. Otherwise, they go back to being blocked. This scheme has an advantage that at the time of completion of the phase, all the threads are ready to move to the next phase without any delay. However, in the worst case, it can result in a large number of redundant operations ($O(\mathtt{n}^2)$), and the redundantly woken up threads may compete with the critical thread for CPU time.



(a) Speedup of `async-finish` over X10 clocks

(b) Speedup of `async-finish` over HJ phasers

FIGURE 2 Speedups for some IMSuite kernels (those with a a significant number of atomic and clock operations) written using only `async-finish` constructs, over their counterparts written using X10 clocks or HJ phasers.

As a consequence of the overheads discussed above, many times we have found that programs written using `Clock-async-finish` (though arguably more readable [4]) have a prohibitively high performance penalty as compared to their counterparts written using only `async-finish` constructs. For example, compared to four kernels from the IMSuite [7] benchmarks that use a significant number of atomic and clock-operations, their async-finish counterparts run on average (geometric mean) 16.46× faster on a 16-core Intel machine (See Figure 2a). A similar observation can be made in the context of the IMSuite [7] kernels and HJ phasers (similar to X10 clocks). As shown in Figure 2b, the same four HJ kernels written with phasers ran on average (geometric mean) 4.25× slower than their async-finish counterparts on the same 16 core Intel system. This shows that even HJ phasers have a similar performance penalty that discourages their use, even though programs written with phasers are more readable.

Prior researchers [8,9] also have recognized the high cost of barriers in task-parallel languages and have proposed newer types of barriers/extensions to address these issues. For example, Shirako et al. [9] fix this issue by using hierarchical barriers for scalable synchronization. Imam and Sarkar [8] use one-shot delimited continuations and event-driven controls to reduce the overheads and improve the efficiency of HJ phasers.

In this paper, we address the efficiency issues of X10 clocks by using a mixed language and runtime based approach. We first propose three desirable properties that an efficient runtime (for languages like X10) should try to satisfy. To satisfy these properties, we introduce new operations on X10 clocks and propose two runtime optimizations to overcome the overheads discussed above. We call this overall scheme to improve the efficiency of lock-step synchronization uClocks. For the above discussed four IMSuite Kernels, we have observed that over varying number of hardware cores (1, 2, 4, 8, and 16) of the Intel system our proposed scheme is able to bring down the performance gap between the async-finish-clocks versions and async-finish versions from 6.75× to 1.26×. This paper focuses on the efficient implementation of lock-step synchronization in the context of shared-memory non-preemptive task-scheduled systems (single place, in X10 parlance).

Though we present uClocks in the context of X10, the set of desirable properties and the proposed optimizations are also meaningful in the context of other task-parallel languages (like HJ and Chapel) that allow lock-step synchronization among tasks (see Section 6, for a further discussion on this topic).

1.2 | Contributions

1. We identify three desirable properties that an efficient task-parallel language and runtime system (with support for lock-step synchronization) should have. We also describe how these properties apply to X10 clocks and why the current implementation does not satisfy these properties.

2. We use the three desirable properties as a basis to design a scheme called uClocks that consists of an extension to X10 clocks, and two runtime optimizations, to improve the efficiency of X10 clocks. Importantly, we prove that the uClocks scheme satisfies the three desirable properties. To the best of our knowledge, we are not aware of any other prior work that gives a proof of such efficiency properties for runtimes, especially those supporting lock-step synchronization.
3. We have implemented uClocks as an extension to the X10 2.6.1 (language + runtime) and evaluated it against the current implementation of X10 clocks. We show that on benchmarks with a large number of atomic and clock-operations, uClocks led to significant speedups: up to 28.18× (geomean 11.39×) on a 64 core AMD system, and up to 22.18× (geomean 5.36×) on a 16-core Intel system.

2 | BACKGROUND

We now briefly describe some features of the X10 language and its XRX (X10 Runtime in X10) runtime, pertinent to this paper. Interested readers may see the language reference manual [10] and the runtime implementation [6], for details. The proposed techniques/optimizations are focussed on a single-hardware-node (typically mapped to a single X10 'place').

2.1 | X10 Parallel Constructs

1. `async S` creates a new asynchronous task that may execute `S` in parallel with the current task.
2. `finish S` waits for all the tasks created in `S`. It gets translated to '`startFinish(); S; stopFinish();`'.
3. `atomic S` enforces mutual exclusion. Here `S` will execute atomically (i.e. in a single step), with respect to all other atomic blocks in the program.
4. `when (c) S` blocks until the condition `c` becomes true, and then `S` will be executed atomically with respect to other atomic-blocks. We refer to the bodies of both `atomic` and `when` statements as atomic-blocks.
5. `Clocks` help realize lock-step synchrony among tasks. A task may register on zero or more clock-objects. Each clock-object has a phase number associated with it. Tasks registered on a clock-object can wait for all other tasks registered on that clock-object to complete their work in the current phase.

Some of the pertinent clock related operations are given below:

1. `Clock.make()` returns a new clock-object and registers the current task on that clock. In the statement `async clocked (list-of-clocks) S`, the child task gets registered to the list of clocks.
2. `c.resume()`: is used to declare (to other tasks registered on clock `c`) that the task has completed the current phase, and other tasks need not wait for it anymore to go to the next phase.
3. `c.advance()`: For a clock `c`, the statement `c.advance()` first makes an implicit call to `resume()`, if it has not already been already called in the current phase. Then, it blocks till all the tasks registered on the clock have executed a `resume()` in the current phase.
4. `Clock.advanceAll()`: This statement calls a `resume()` on all the clocks that the task is registered on, followed by an `advance()` on all the clocks the task is registered on.
5. `c.drop()`: This statement de-registers the current task from the clock `c`, and the task is no longer considered for any synchronization on `c`.

2.2 | X10 Runtime Implementation

Before executing the `main` function, the X10 runtime creates X10_NTHREADS (an environment variable) number of worker threads (or, threads in short). Typically it is set to the number of cores on the processor. The X10 runtime uses a work-stealing scheduler [11], where each thread has a deque (short for double-ended queue) of tasks to execute.

At any point of time a thread can be in one of the following two states:

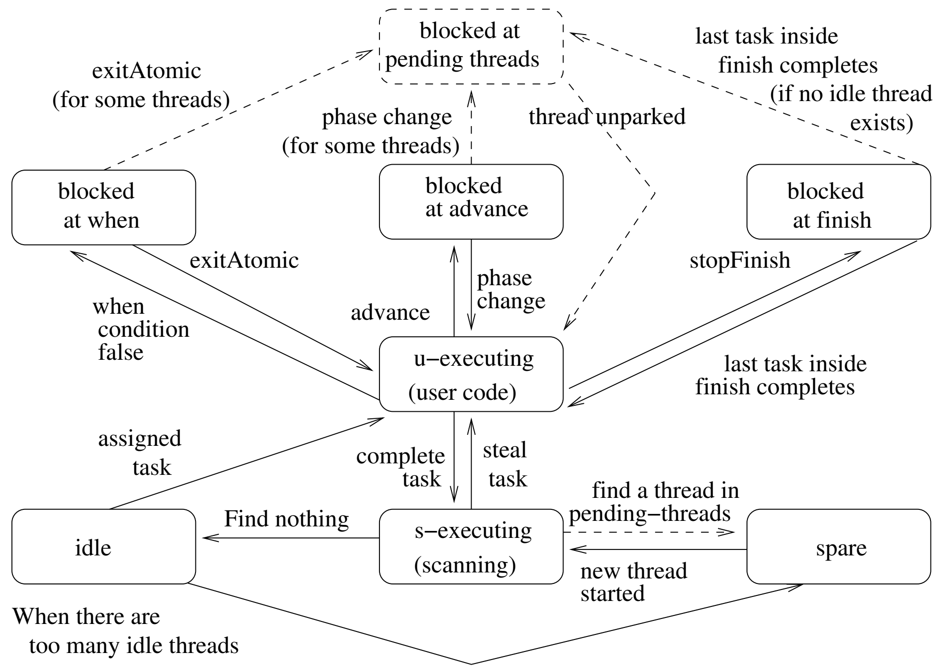


FIGURE 3 State transition at runtime. Solid lines: state transition for the current X10 implementation. Dashed lines: added states/transitions for uClocks – discussed in Section 4.2.

1. *executing* - the thread is executing some code and could be in one of the two sub-states:
   (a) *u-executing* - the thread is executing some user code.
   (b) *s-executing* - the thread is scanning deques to execute/steal tasks.
2. *parked* - the thread is not executing any code (not considered for scheduling). A parked thread could be in one of the following sub-states:
   (a) *idle* - an idle-thread is ready for work and can be assigned a new task if it comes up.
   (b) *blocked* - a blocked-thread is waiting at an `advance`, `finish`, or `when`.
   (c) *spare* - to avoid the overheads of repeated thread creation/termination operations, the runtime marks a thread as "spare", instead of killing it. And in future, if a new thread is to be created, a spare thread is "re-started", if available. Otherwise, a new thread is created.

At startup, the main thread is in executing-state and the rest are in idle-state. For a quick glance, we show the thread-state transition in Figure 3.

We now briefly discuss how the X10 runtime handles certain pertinent runtime events. On encountering an `async`-statement, the runtime checks if there are any idle threads. If there is at least one idle thread, the task is assigned to it. Otherwise, the new task is put on the deque of the current thread. In either case, the current thread continues to execute the statement after the `async`. On completion of a task, the thread starts executing the next task from its deque. If the deque is empty, it will attempt to steal a task from other threads. If unsuccessful, it will mark itself as an idle thread.

On encountering the `stopFinish()`, the thread will park itself until all the tasks created in the body `S` of the `finish` have terminated. The thread to complete the last available task in `S`, unparks the thread parked in the `stopFinish()` to continue the execution beyond the `finish` statement.

To implement the `atomic` construct, the runtime maintains a shared lock (`amLock`). The statement '`atomic S`' gets compiled into '`enterAtomic(); S; exitAtomic();`'. The function `enterAtomic` blocks till it acquires `amLock` and `exitAtomic` releases `amLock`. Further, `exitAtomic` also unparks all tasks parked in a `when` statement.

```
1 enterAtomic();
2 while cond ≠ true do
3 |  amLock.unlock();
4 |  park(); /* may start a new thread */
5 |  amLock.lock();
6 S;
7 exitAtomic();
```

FIGURE 4 Compiling `when(cond)S`

```
def resume()
    enterAtomic();
    alive = alive − 1 ;
    if alive==0 then
        alive = count;
        current clock phase++;
    exitAtomic();
```

FIGURE 5 Implementation of `resume`

```
def advance()
    call resume() if not already called in this phase;
    ph = current phase of the task;
    when (ph < current clock phase);
    Increment the phase of the current task;
```

FIGURE 6 Implementation of `advance`

Figure 4 shows the translation of the `when` statement by the X10 compiler. This code also uses `amLock`, discussed above. If `cond` is false, the thread releases the lock and parks. The thread is unparked by another thread executing `exitAtomic`.

We draw attention to 2 subtle points in this implementation:

(i) Every `park` operation is accompanied by the starting of a thread, in order to ensure that the number of actively running threads remains the same, and the computation makes progress.

(ii) When a thread is blocked, it cannot execute any other task as it holds the context to execute the code following the "blocking" statement. Thus in parts of the program with clocks, there is a one-to-one mapping between the tasks and threads, and we use these terms interchangeably in such contexts.

2.3 | X10 Clocks Implementation

Each clock-object includes three counters: (i) `phase` records the current phase number of the clock. (ii) `count` records the number of tasks registered on the clock. (iii) `alive` records the number of tasks registered on the clock which have not executed `resume()` in the current phase.

We now discuss the implementation of the `resume` and `advance` functions. Figure 5 shows the pseudo code for `resume()`. This code (executed atomically) decrements the `alive` count by 1 and increments the clock phase if rest of the tasks registered on this clock have executed `resume`.

Figure 6 shows the pseudo code for `advance()`. It first calls `resume()`, if the current task has not already called `resume()` explicitly. Next, the thread blocks in the `when` statement until the phase of the clock changes (done by the last task executing `resume`).

3 | DESIRABLE PROPERTIES FOR AN EFFICIENT RUNTIME

One of the critical overheads incurred by the runtimes of task-parallel languages that support fine-grain synchronization of tasks, is related to the inefficient implementation of synchronization primitives. Additionally, this can lead to a situation where a large number of active threads ($\gg$ # available cores) are present – leads to performance deterioration. Based on our observation of

the implementation of many task-parallel languages like X10, HJ, Chapel, Java and so on, and general efficiency considerations, we first describe some of the properties that are desirable in such runtimes, so as to reduce these overheads. We then discuss the conformance of the current X10 runtime, which is the focus of our paper, to these properties.

3.1 | Desirable Properties to hold during program execution

1. (*DP1*) *Instances of lock-unrelated parallel constructs should not share locks.* We define two instances of executions of one or more parallel constructs to be *lock-related* if (1) their underlying semantics require that they must access some shared runtime-internal data, and (2) these accesses must be performed in a mutually-exclusive manner to avoid data-race. For example, in X10, all the instances of `atomic` constructs are lock-related; their execution must acquire a common lock to ensure mutual-exclusion as guaranteed by the X10 language. However, an instance of any `atomic` construct and any clock-related operation are lock-unrelated; the clock-related operations may run in parallel with the `atomic` blocks. Using shared locks in the implementation of lock-unrelated constructs may lead to unnecessary serial executions. In addition, such usages may lead to additional overheads (see Section 3.2).
2. (*DP2*) *At runtime, the user must specify a value for* `maxThreads` *(*> 0*), signifying the desired number of worker threads. And during the execution of the program, the number of actively running threads should be as close to the value of* `maxThreads` *as possible, but no more.* Typically, users set the value of `maxThreads` to the number of CPU cores on the system (to ensure that each thread is more or less mapped to one core). If the #active-threads < #cores, we are not fully utilizing the available resources, and if we have too many threads, then the overheads due to thread context switching negate the benefits of parallelism.
3. (*DP3*) *At each barrier, the total number of operations executed (by all the tasks) should be proportional to the number of tasks registered on the barrier*. At each barrier, the set of performed operations must ensure that if there are tasks with pending work in the current phase, then the other threads must wait at the barrier. Otherwise, all the tasks should be informed about the completion of the current phase. DP3 would guarantee that the average number of operations performed (per thread) at a barrier is a constant and is independent of the number of tasks, number of threads, or any other such factors.

3.2 | Conformance of X10 to the desirable properties.

In this section, we look at how the above discussed desirable properties apply to the current X10 runtime (XRX). Interested reader may refer to Section 8, for a brief discussion on how other task-parallel languages conform to the desirable properties.

3.2.1 | X10 and DP1

Multiple instances of the `finish` statement are lock-unrelated to each other and multiple instances of `atomic` and `when` constructs are lock-related. In the X10 runtime, each instance of the `finish` construct uses a separate lock and the implementation of `atomic` and `when` constructs share a common lock across all the instances (see Section 2.3). Thus, X10 programs that use only finish-async-when-atomic constructs to harness parallelism conform to DP1.

The implementation of the clock-related operations, such as `advance()`, `resume()`, and so on, use the X10 `atomic` and `when` constructs underneath. An important point to note is that these clock-related operations on different clock-objects are lock-unrelated, but since atomic-blocks are used to implement the above clock-related operations, it leads to sharing of locks between lock-unrelated operations – a violation of DP1. Similarly, the atomic-blocks in the user-code share locks with the above atomic-blocks invoked by the clock-related operations – again, a violation of DP1.

Note: the violation of DP1, by the X10 runtime, causes overheads even beyond those resulting from lock contention. For example, for the code shown in Figure 7 after every execution of the atomic-block (in the while loop), the X10 runtime unparks (see Section 2) every thread parked on a `when` statement. Since the `advance` construct is implemented using the `when` construct, if a thread is parked at the `advance` construct (unrelated to the operations performed inside the atomic-block) it gets unparked unnecessarily, only to go back to park mode again, till the other thread reaches the `advance()`. Similar issues can be observed in the presence of `when` statements in user-code. Such dependencies between unrelated constructs are purely an artifact of the implementation and should be avoided.

```
async {
    c = Clock.make();
    clocked async (c) {
        S1;
        c.advance();
        S2;
    }
    S3;
    c.advance();
    S4;
}
while(i<n){
    atomic{ ... }
}
```

FIGURE 7 Example to show how DP1 can be violated

Advice: To ensure that DP1 is not violated: 1) we need to make sure that the runtime does not use atomic-blocks; note: the runtime does not share any variables with the atomic-blocks in the user-code. Instead, the required mutual exclusion among the operations performed in the runtime should be realized using a separate (set of) lock(s). 2) Further, all the clock-objects can have their own locks, as they are independent of each other.

3.2.2 | X10 and DP2

In the current X10 Runtime, the user specifies an environment variable X10_NTHREADS, indicative of `maxThreads`. When the last task registered on a clock executes `resume()` (as shown in Figure 5), the `phase` variable of the clock is incremented. On executing `exitAtomic()`, all the tasks registered on the clock move to the next phase simultaneously and all their threads are in 'active' state. Hence, the number of active threads is O(Number of tasks), which typically is much larger than X10_NTHREADS – hence, a violation of DP2. A similar violation is observed during the execution of `when` statements (see Figure 4), where the runtime creates an additional thread, every time a thread is blocked on a `when` construct. When any other thread executes the `exitAtomic()` at the end of any `atomic`- or `when`- construct, all the threads (O(#num-tasks)) parked at the `when` construct are unparked – a violation.

Note: the DP2 second requirement – maximizing the number of active threads – is satisfied by the current X10 runtime.

Advice: Thus to ensure that DP2 is not violated, (1) the number of active threads ($\leq$ `maxThreads`) should be maximized and (2) the maximum number of threads unparked during the phase change operation, should be bound by the expression (`maxThreads`−number of active threads).

3.2.3 | X10 and DP3

In the current X10 implementation, the `resume` function ends with an `exitAtomic()` function (see Figure 5), which unparks every thread parked in the `when` construct (called from an `advance()` or explicitly). Each such unparked thread blocked at the `advance()` barrier checks if the phase has changed, or else it goes back to park mode. Thus, if there are p tasks registered on a clock, in every phase, in the worst-case scenario, the qth `resume()` triggers $(q - 1)$ unpark and park operations – worst case $O(p)$. Thus the total number of operations performed, at the barrier, across all the p tasks is $O(p^2)$ - a violation of DP3.

A consequence of unnecessarily unparking these parked threads is that they compete with the critical thread(s) for CPU time, which in turn may lead to deterioration of performance.

Note that unless all the tasks (one per thread) have completed the phase, the `when` condition, in the `advance()`, will not become true and hence all the threads need not be unparked every time a task executes a `resume()` operation.

Advice: To ensure that DP3 is not violated, either (i) only the last instance of the `resume()` operation of a phase should unpark all the tasks (threads) parked at the `advance()` – *Lazy* scheme, or (ii) the threads are eagerly unparked but they proceed to the next phase, without having to re-park (or busy wait) – *Eager* scheme. The latter is possible when the threads are waiting for

tasks executing small jobs: even if the parked threads get unparked, before they can even check the condition for phase-change the "small" tasks complete the phase.

4 | UCLOCKS: AN EXTENSION

We use the three desirable properties as a basis to design a scheme called uClocks to improve the efficiency of X10 clocks. It consists of (i) an extension to X10 clocks that helps satisfy DP3 by leading to an efficient implementation of the `resume` and `advance` operations. (ii) Two runtime optimizations to help satisfy DP1 and DP2. We now discuss the extension to X10 clocks and discuss the proposed optimizations in Section 4.2.

4.1 | Extending Clock operations for Efficiency

4.1.1 | uClocks - Motivation for extending the syntax

As discussed in Section 2, the current X10 runtime (implementation of clocks), unparks every parked thread at each `resume()` call. Each such unparked thread (when scheduled by the underlying OS) checks if there is a phase change, else it goes back to park mode. We refer to such a scheme of eagerly unparking every thread as the `Eager` scheme. An advantage of such a scheme is that when the last task finishes the current phase, in the best case scenario, the other threads are already unparked and are ready to execute in the next phase – no delay incurred to unpark, after the phase completion. Further, in tasks without much load imbalance, every such eagerly unparked thread may end up checking for phase change, only after the phase has actually changed (no further parking/unparking) – satisfies DP3, as every thread performs at most one park/unpark operations. We call such scenarios as *eager-friendly*.

However, such a scheme may lead to large overheads, due to DP3 violation, when the tasks are more in number than the cores (a common case), and have significant load imbalance.

As discussed in Section 3.2.3, another way to satisfy DP3, is to unpark every thread parked in an `advance()` only when it is ready to go into the next phase. Hence, only the last task to execute the `resume()` for the current phase, will unpark all the parked threads mapped to tasks registered on that clock. We refer to such a scheme as the `Lazy` scheme. In contrast to the `Eager` scheme, even though the `Lazy` scheme always satisfies DP3, it incurs additional delay that results from unparking the threads after the completion of the phase. In the presence of unbalanced workloads, this delay usually gets masked by the gains resulting from avoiding the overheads due to the scheduling of the prematurely unparked threads.

We have observed that programs may contain code that has phases with both balanced and unbalanced loads. We have observed that using a single synchronization scheme (all `Lazy`, or all `Eager`) for all the cases is not optimal. Further, considering the individual advantages and disadvantages of the `Lazy` and `Eager` schemes, and the difficulty in statically/dynamically estimating phases of tasks with significant load-imbalance, we present an extension to X10 clocks to let the programmer take advantage of both the schemes depending on the requirement (more load-imbalance - use `Lazy` scheme). This motivation of our proposed extension is similar to that of the well understood `OMP_WAIT_POLICY` of OpenMP[12].

4.1.2 | uClocks - Extended Syntax

We now propose an extension to X10 clocks that allows the user to specify the synchronization scheme (`Eager` or `Lazy`), at each synchronization point. In place of the default `resume` and `advance` functions our proposed extension admits four new operations: `resumeEager()`, `resumeLazy()`, `advanceEager()`, and `advanceLazy()`. The latter two functions, call their corresponding variants of `resume`. By explicitly naming the synchronization primitives, we encourage the programmers to explicitly use the specific synchronization scheme at each synchronization point. We discuss the implementation details of the proposed extensions in Section 4.2.3.

4.2 | Runtime Optimizations for uClocks

We now describe two optimizations for the X10 runtime (to satisfy DP1 and DP2). These optimizations also pave the way for an efficient implementation for the extension to X10 clocks discussed in Section 4.1.

```
1  def advance()
2  |  Call resume() if not already called in this phase;
3  |  ph = current phase of the task;
4  |  clLock.lock() ;
5  |  while ph==current clock phase do
6  |  |  record current thread in clk-blocked-threads ;
7  |  |  clLock.unlock() ;
8  |  |  park(); // Don't busy wait.
9  |  |  clLock.lock() ;
10 |  clLock.unlock() ;
11 |  Increment the phase of the current task.
```

FIGURE 8 Implementation of uClocks-Opt1 `advance`

```
1  def resume()
2  |  clLock.lock() ;
3  |  alive = alive − 1 ;
4  |  if alive==0 then
5  |  |  alive = count ;
6  |  |  Increment the phase of the current clock.
7  |  unpark threads in clk-blocked-threads ;
8  |  clLock.unlock() ;
```

FIGURE 9 Implementation of uClocks-Opt1 `resume`

4.2.1 | uClocks-Opt1 - Optimization to satisfy DP1

As discussed in Section 3.2.1, the `resume` and `advance` operations use X10 `atomic` and `when` constructs underneath. We use the given advice in Section 3.2.1 and (i) define a separate lock with each clock-object (field named `clLock`), (ii) use these locks instead of `atomic` and `when` constructs. Recall (see Section 2) that the `when` operation is a blocking operation, where a number of tasks may be waiting for a certain condition to be true (for example, if all the tasks have completed a phase). The implementation of the `when` construct, records these tasks in an internal list. To mimic this behavior, we maintain a new list called `clk-blocked-threads` per clock, to record all the threads that are blocked at the corresponding `advance` operation.

Figure. 8 shows the proposed `advance` function. In contrast to the code shown in Figure. 6, Lines 4-10, mimic the behavior of the substituted `when` statement.

The implementation of `resume` (see Figure. 9) is similar to that shown in Figure. 5, except that the `enterAtomic()` and `exitAtomic()` calls are replaced by invocations to `clLock.lock()` and `clLock.unlock()`, respectively. For each clock-object, its `clLock` field, protects accesses to all its shared fields. Further, before invoking `clLock.unlock`, to mimic the behavior of `exitAtomic`, we unpark all the threads present in `clk-blocked-threads`.

4.2.2 | uClocks-Opt2 - Optimization to satisfy DP2

To satisfy DP2, we use the advice given in Section 2 and during all the Clock related operations, we maintain the following invariant on idle threads (say, IT=number of idle threads) and executing threads (say, ET=number of executing threads); see Section 2 for their definitions.

$$\text{IT} + \text{ET} = \text{X10_NTHREADS} \tag{1}$$

This invariant helps us maintain $\text{ET} \leq \text{X10_NTHREADS}$. Since the current runtime already minimizes the number of idle threads, Eq (1) does not lead to any loss in parallelism.

Maintaining the above invariant would lead to a situation where even after all the tasks have completed the current phase, only some of the tasks start executing in the next phase, while others are waiting in park mode; we call the threads mapped to these waiting tasks as *pending*. We record these pending threads in a new list called `pending-threads`.

```
1  def resume()
2      clLock.lock() ;
3      alive = alive − 1 ;
4      if alive==0 then
5          alive = count ;
6          Increment the phase of the current clock. ;
7      i = min(no of tasks registered on clock, IT) ;
8      mark i idle threads as "spare";
9      unpark i threads from clk-blocked-threads ;
10     Add the remaining threads of clk-blocked-threads to pending-threads ;
11     clLock.unlock() ;
```

FIGURE 10 Implementation of uClocks-Opt2 `resume`

To enforce the above invariant, we modify the `resume` function, as shown in Figure. 10. Here, at Lines 7-9, a task executing `resume`, does not wake up all the tasks parked at the corresponding `advance`. Instead, it only unparks at most as many threads as the number of idle threads. These idle threads are marked as "spare" (see Section 2) – this step reduces the number of idle threads, to match the increase in ET at Line 9. At Line 10, the remaining tasks blocked on that clock get added to the `pending-threads` list, and will get unparked later on, when an executing thread is about park.

The code for `advance()` (not shown) is exactly the same as that shown in Figure. 8, except that before invoking the park method, the thread first attempts to unpark one thread from `pending-threads` (if size > 0, that is). The corner case of all other threads having dropped the clock and a single thread is 'pending' is handled separately, by checking `pending-threads`, when a thread has no tasks to steal.

To ensure that DP2 is satisfied in the presence of the `when` construct, we modify its implementation (given in Section 7) so that the thread reaching the `exitAtomic()` unparks k threads parked at `when` constructs, where k = min (IT, number-of-threads-parked-at-when-constructs). The remaining parked threads are moved to `pending-threads`. These threads in `pending-threads` will get unparked at a later point in time, when another thread is parking itself (in the `when` / `advance`, or `stopFinish()`).

Similarly, within a `finish`, when the last task completes, the executing thread T marks an idle thread as spare and unparks the thread waiting at `stopFinish`. If idle threads are unavailable, then the T becomes a spare thread itself.

4.2.3 | Implementation of the proposed extension

Figures 11 and 12 show our proposed implementation for the `resumeLazy` and `advanceLazy` functions, respectively. We discuss these implementations in conjunction with the proposed optimizations in Sections 4.2.1 and 4.2.2. In contrast to the implementation of `resume` shown in Figure. 10, in the implementation of `resumeLazy`, the unparking of threads from `clk-blocked-threads` is done inside the `if` statement – the unparking is done lazily only once, at the end of the phase. Similarly, in contrast to the `advance()` implementation discussed in Section 4.2.2 (which in turn, is a minor modification of the code shown in Figure. 8), in the implementation of `advanceLazy` the `while` statement is replaced with an `if` statement, because every thread will now park-unpark only once inside the `advance()`.

The implementations of `resumeEager` and `advanceEager` functions are exactly the same as the `resume` and `advance` functions discussed in Section 4.2.2.

5 | PROVING PROPERTIES OF UCLOCKS

To establish the confidence on the proposed uClocks scheme, we now prove that uClocks satisfy the desirable properties discussed in Section 3, and briefly discuss the semantic equivalence of uClocks and X10 clocks. To the best of our knowledge, we are not aware of any other prior work that gives a proof of such efficiency properties for runtimes, especially those supporting lock-step synchronization.

```
1  def resumeLazy()
2      clLock.lock() ;
3      alive = alive − 1 ;
4      if alive==0 then
5          alive = count ;
6          Increment the phase of the current clock ;
7          i = min(no of tasks registered on clock, IT) ;
8          mark i idle threads as "spare";
9          unpark i threads from clk-blocked-threads ;
10         Add the remaining threads of clk-blocked-threads to pending-threads ;
11     clLock.unlock() ;
```

FIGURE 11 Implementation of `resumeLazy`.

```
1  def advanceLazy()
2      Call resumeLazy() if not already called in this phase;
3      ph = current phase of the task;
4      clLock.lock() ;
5      if ph==current clock phase then
6          Add thread to clk-blocked-threads ;
7          clLock.unlock() ;
8          park() ;
9      Increment the phase of the current task.
```

FIGURE 12 Implementation of `advanceLazy`.

5.1 | uClocks and DP1

Each instance of the `finish` construct and each individual clock-object now have an individual lock. Each instance of the `atomic` and `when` constructs share the common atomic lock, since they are lock-related. Hence, we satisfy the condition that instances of lock-unrelated parallel construct have separate locks – satisfies DP1.

5.2 | uClocks and DP2

We first prove that Eq (1) holds by using induction on the runtime-events that change the number/status of threads.

Base case: At startup, the Runtime has exactly one executing-thread and (X10_NTHREADS − 1) idle-threads. Here the Eq (1) holds.

Induction step: Say Eq (1) holds at some point during the execution. That is, $\text{IT} = n - \text{ET}$, where n = X10_NTHREADS. There are nine possible cases, depending on which runtime-event (that may modify ET or IT) occurs next. We now describe each of these cases and show their impact on the value of the expression $\text{ET} + \text{IT}$. We use the notation $n \uparrow^k$ (or $n \downarrow_k$) to indicate that value of n increases (or decreases) by k.

1) Event - new task creation: There are two sub-cases:

i. Currently, $\text{IT} = 0$. As discussed in Section 2, the task is put on the deque of the current thread, and no parking or unparking of threads is done (no change to IT or ET).

ii. Currently, $\text{IT} \neq 0$. As discussed in Section 2, the new task is assigned to one of the idle threads and that idle thread gets unparked ($\text{IT} \downarrow_1$ and $\text{ET} \uparrow^1$).

2) Event - task termination: There are two sub-cases:

i. Last task within a `finish` completes: As explained in Section 4.2.2, when the last task completes, (a) it either converts one idle thread to spare ($\text{IT} \downarrow_1$) or marks itself as spare ($\text{ET} \downarrow_1$), and (b) unparks the thread waiting at the `stopFinish` ($\text{ET} \uparrow^1$).

ii. Otherwise: the thread starts scanning deques to execute/steal tasks. This does not impact ET and IT.

3) Event - `advance`*:* The thread executing the `advance` is parked (ET $\downarrow_1$) and another thread is unparked (ET $\uparrow^1$).

4) Event - `resumeEager`*:* In Figure 10, the reduction in IT at Line 8 is compensated by the increase in ET at Line 9.

5) Event - `resumeLazy`*:* In Figure 11, the reduction in IT at Line 8 is compensated by the increase in ET at Line 9.

6) Event - `when`*:* (see Figure 4) If the associate condition *e* evaluates to true then the thread simply continues executing (status of no thread changes). Otherwise, the current thread parks (ET $\downarrow_1$), and another thread is unparked (ET $\uparrow^1$).

7) Event - `exitAtomic`*:* As explained in Section 4.2.2, at the `exitAtomic` k threads parked at `when` statements are unparked, and k idle threads are converted to spare. (ET $\uparrow^k$ and IT $\downarrow_k$), where $k = \min$ (IT, number-of-threads-parked-at-when-constructs). Note: The `enterAtomic()` event does not impact ET and IT.

8) Event - `stopFinish`*:* As discussed in Section 2, the thread parks at `stopFinish` (ET $\downarrow_1$), and unparks another thread (ET $\uparrow^1$). Note: The `startFinish()` event does not impact ET and IT.

9) Event - a thread T is scanning for work: There are two sub-cases.

i. `pending-threads` is non-empty. As discussed in Section 2, T marks itself as 'spare' and parks itself (ET $\downarrow_1$), and unparks a thread from `pending-threads` (ET $\uparrow^1$).

ii. `pending-threads` is empty. If T is unsuccessful in finding a new task to execute then it marks itself as idle (ET $\downarrow_1$ and IT $\uparrow^1$). Otherwise, it executes the found task – no change to ET or IT.

In all the nine cases, since Eq (1) held before the event, it will hold even after the event as the value of IT + ET remains unchanged. □

As discussed in Section 3.2.2, besides satisfying equation 1, DP2 requires that ET be kept as high as possible. The X10 runtime already ensures that IT is minimized. This fact in conjunction with Eq (1) ensures that DP2 is satisfied.

5.3 | Deadlock due to reduction in ET

The previous section shows that DP2 is satisfied by proving that Eq (1) holds. We now present an argument to reason about the absence of deadlocks due to uClocks; that is, during execution ET is always a positive number (> 0). Similar to the argument in Section 5.2, we prove the current argument by using induction on the runtime-events that change the number/status of threads.

Base case: At startup, the Runtime has exactly one executing-thread and (X10_NTHREADS – 1) idle-threads. Here ET > 0 holds.

Induction step: Say ET > 0 holds at some point during the execution. As discussed in Section 5.2, there are nine possible runtime-events that can occur and impact the value of ET. Of these nine cases, the first eight either increase the value of ET, or do not modify it. Only the ninth case decreases ET and is detailed below.

9) Event - a thread T is scanning for work: There are two sub-cases.

i. `pending-threads` is non-empty. As discussed in subsection 5.2, this case does not modify ET

ii. `pending-threads` is empty. There are 2 further subcases.

- If T is successful in finding a new task to execute (either from its deque or from another thread's deque), it simply executes that task, and there is no change in ET.
- If T is unsuccessful in finding a task to execute, it marks itself as idle (ET $\downarrow_1$ and IT $\uparrow^1$). There are further 3 subcases
  - If ET > 1 before the thread marks itself as idle: Since ET reduces by 1 and even after this step ET > 0.
  - If ET = 1 before the thread marks itself as idle: the thread is the last executing thread. We already are in the subcase where there are no new tasks to execute, and `pending-threads` is empty (no tasks stuck in `advance`, `when` or `stopFinish`). This means that there are no more instructions left to execute, and the program will end after after this step. That is ET remained a positive number, during program execution.
  - ET < 1 before the thread marks itself as idle: a contradiction to the assumption that ET > 0 before the step.

Since none of the nine runtime-events can violate the condition of ET > 0 during program execution, and the program execution starts off with ET = 1, we can ensure that the condition ET > 0 continues to hold during program execution. □

5.4 | uClocks and DP3

We now show the conformance to DP3 by discussing the number of operations performed in the contexts of `resumeEager` in eager-friendly conditions (Section 4.1.1), and `resumeLazy`.

Case 1: `resumeEager()` in eager-friendly scenarios - Assuming that `resumeEager()` gets called only in eager-friendly scenarios, each thread (one per task) gets unparked and parked just once, at the `advance`, and in the `advance` function, it executes a constant number of operations. Since every thread t unparks only 1 thread in `resumeEager()`, t executes a constant number of operations. Thus, the total work done by k tasks in `resumeEager` (and `advanceEager`) functions is $O(k)$.

Case 2: `resumeLazy()` - Assume that there are k tasks registered on the clock. The algorithm in Figure 11, ensures that each task invokes a `resume` (and `advance`) operation only once per phase. Each call to `resumeLazy` (except the last one in the phase) involves just $O(1)$ operations – total $O(k)$.

The last task to invoke `resumeLazy()`, in the phase, performs a few additional operations; shown in lines 5–10, in Figure 11. Since, IT less than X10_NTHREADS (Eq (1)), the number of operations in any of the statements is at most MAX(k, X10_NTHREADS). Since, in general k is greater than X10_NTHREADS, the work done by the last task is $O(k)$.

The number of additional operations (beyond that of `resumeLazy`) performed by `advanceLazy` functions is $O(1)$. Thus, the total work done by k tasks in `resumeLazy` (and `advanceLazy`) functions is $O(k)$.

Thus in both cases, DP3 is satisfied. □

5.5 | Semantic equivalence of uClocks and X10 clocks, and deadlock gurantees

(Brief sketch) uClocks Opt1 (Section 4.2.1) simply replaces the `amLock` lock, with individual locks of the same type, for every clock, and the functionality of the `atomic` and `when` constructs are simply replicated using these individual locks. Since clock-objects are independent of each other, this change maintains the equivalence.

uClocks Opt2 (Section 4.2.2) has the modification that the last `resume` unparks only IT threads parked at the `advance`. Each of the unparked threads will either reach a barrier, in which case they will unpark a thread from `pending-threads`, or they will complete their task, scan for work, and finally end up unparking a thread from the `pending-threads` list. Hence, in either case, the parked threads at the `pending-threads` list eventually get unparked and the overall computation always makes progress. Importantly, for any program, the schedule of threads realized in the uClocks scheme is one of the permissible schedules as observed when the program is run using X10 clocks.

For uClocks Extension to clocks (Section 4.2.3), the `Eager` scheme is the one followed in the original X10 Clocks implementation, and hence is trivially equivalent. The `Lazy` scheme simply postpones the unparking of threads parked at the `advance` to the last `resume`, and since all tasks can only move to the next phase after the last `resume`, the equivalence holds, in this case as well.

Since the two optimizations and language extension do not change the semantics of X10 clocks, we can say that the uClocks scheme on the whole preserves the semantics of X10 clocks. Further, since the deadlock semantics are a part of the X10 clocks semantics, a preservation of the semantics of X10 clocks implies a preservation of the deadlock semantics of X10 clocks. In other words, a program using uClocks will deadlock if and only if the program using X10 clocks deadlock. Interested reader may refer to X10 specification[10] to understand the cases where the usage of clocks may lead to deadlocks.

6 | BARRIERS IN OTHER TASK-PARALLEL LANGUAGES

In this section, we describe the barrier constructs present in other task-parallel languages like HJ[2], Chapel[3], and Java[13]. The proposed desirable properties (introduced in Section 3.1) are applicable to all these three languages because they all support task-level lock-step synchronization. An analysis of their barrier implementations shows that they do not satisfy all the desirable properties (summarized in Figure 13). Though X10 does not satisfy DP1 (see Section 3.2), languages like HJ, Chapel and Java do satisfy DP1. None of the languages fully satisfy DP2 or DP3; Java (in the java.util.concurrent.Phasers package) satisfies DP3 partially – it supports only the 'Lazy' scheme (discussed in Section 3.2) – which can lead to inefficiencies sometimes.

In this paper, we used the X10 language as the vehicle to discuss the conformance issues of the desirable properties in detail, propose schemes to ensure that these properties are satisfied and study the impact of the proposed schemes. The importance of satisfying these properties is highlighted in the significant performance improvement we achieve for X10 (see Section 7). Since HJ, Chapel and Java have barrier constructs which are very similar to those in X10 clocks, and their runtime

Desirable Property	X10 clocks	HJ Phasers	Chapel Barriers	Java Phasers
DP1	Not Satisfied	Satisfied	Satisfied	Satisfied
DP2	Not Satisfied	Not Satisfied	Not Satisfied	Not Satisfied
DP3	Not Satisfied	Not Satisfied	Not Satisfied	Partly satisfied

FIGURE 13 Conformance of various task-parallel languages to the Desirable Properties

implementations do not satisfy at least some of the desirable properties, we believe that similar schemes can be successfully extended to these task-parallel languages; the actual impact may vary depending on the specific runtime internals. A full fledged implementation/evaluation for these languages and their runtime-internals is left as future work.

In additions to the clock related functionalities discussed in this paper, lock-step synchronization in some of these languages supports additional features. For example, HJ Phasers and Chapel Barriers get implicitly dropped when the barrier object goes out of scope. HJ Phasers can, apart from the default mode of both signaling and waiting, be run in two special modes: signal only or wait only. HJ Phasers also support a variation of the `next` statement (counterpart of X10 `advance`) called the `next single Stmt`, which executes a single instance of `Stmt` when performing the phase transition. Phasers in HJ and Java have an additional feature wherein phasers may be organized in a tree hierarchy to reduce lock contention[9]. The ideas proposed in this paper are also applicable in the context of these additional features, all of which require accessing some shared resources (work-lists, queues, locks, and so on).

Some older explicitly-parallel programming languages like Unified Parallel C[14], Split-C[15], Co-array Fortran[16] and Titanium[17] have support for barriers, but these languages require the programmer to express parallelism directly in terms of threads and not using the abstraction of tasks. OpenMP[12] is a task-parallel language that supports synchronization of worker-threads (but not tasks) using barriers. There has been prior work[18,19,20] on introducing task barriers and phaser type barriers among threads for OpenMP. Expressing parallelism directly using threads (as in Unified Parallel C, Split-C, and so on) or programming thread-barriers in a task-parallel language (as in OpenMP), arguably makes it hard to write code that efficiently utilizes the multi-core architecture, and hence we do not focus on this class of languages in this paper.

7 | IMPLEMENTATION AND EVALUATION

We extended the X10 2.6.1 language with our proposed extension and implemented it in the XRX runtime, along with optimizations proposed in Section 4.2. Instructions for downloading and running the software, and repeating the experiments from this section, are specified in the appendix. The X10 compiler supports the generation of code to two target languages: Java and C++. Both of them use the same XRX runtime. For brevity, we detail the performance measurements for only one backend (Java); the results with the C++ backend are similar (briefly summarized at the end of this section). The goal of our evaluation is to study the impact of the proposed extension to clocks and optimizations, and establish the importance of the desirable properties (Section 3).

While the proposed desirable-properties are relevant both in the context of single-node and multi-node systems, in the context of X10 (and other similar languages), their impact/gains are only from individual nodes. Hence we present a single-place based evaluation.

We performed the evaluations on two shared memory systems: a 16 core Intel system (2 Intel E5-2670 2.6GHz processors, 8 cores/processor, 64GB RAM), and a 64 core AMD system (4 AMD Abu Dhabi 6376 processors, 16 cores/processor, 512GB RAM). We ran the benchmarks for varying number of cores (in powers of two); for running on a system with k cores, we set X10_NTHREADS to k. We take inspiration from the insightful paper of Georges et al.[21] and report numbers as an average over 30 runs to account for any runtime fluctuations.

We performed the evaluation using the iterative kernels of IMSuite[7]. These kernels (listed in Figure 14) encode some of the popular distributed algorithms in use: breadth first search (BF and DST), committee creation (KC), leader election (DP, HS, and LCR), maximal independent set (MIS), minimum spanning tree (MST), byzantine consensus (BY), dominating set (DS) and vertex coloring (VC). For all the benchmark kernels, we use input size = 512 nodes, except for BY and DS (takes too long a time), for which we set input size = 128 nodes.

	LOC	I/P Size	#advances	#atomics	#advances per sec	#atomics per sec
KC^L	562	512	9216	107182	318	3696
HS^L	500	512	1046528	0	2072	0
LCR^L	322	512	262144	0	1273	0
BY^L	556	128	4352	537064	14	1684
BF^E	370	512	1536	9214	24	146
DST^E	566	512	5120	14694	35	100
MIS^E	421	512	6144	44320	23	164
MST^E	923	512	81408	7246	72	6
VC^E	464	512	2048	0	55	0
DS^L	612	128	5232	599	77	9
DP^{LE}	474	512	16896	102772	24	147

FIGURE 14 Statistics for the benchmark kernels used. The superscript on the benchmark denotes the type of barriers used: Lazy (L), Eager (E) or both (LE).

Figure 14 shows some statistics about the chosen kernels. Static: lines of code and the type of synchronization used. Dynamic: used input, and the number of sync-ops (advance-operations and atomic-operations) and sync-ops per second. The latter is computed using the execution time of the default X10 runtime, on the Intel system using a single core.

We partition the kernels into two groups based on whether the kernels have a large number of sync-ops per second at runtime (top part – High-sync-op kernels) or not (bottom part – Low-sync-op kernels), and present our evaluation for both the groups separately.

The intuition behind using sync-ops per second as the basis of the split is that uClocks only optimizes synchronization operations, and naturally if synchronization operations do not a occupy a significant proportion of the execution time in a benchmark, then the effect of the optimization will not be visible in that benchmark. Sync-ops per second is a proxy for the proportion of the time spent by a program in executing synchronization operations, and hence compared to the Low-sync-op kernels, the High-sync-op kernels can be expected to benefit more from uClocks.

For each of the benchmarks, depending on the context in which the `advance` methods are called, we manually replaced the calls individually to their eager or lazy variant to obtain a code that satisfies DP3.

7.1 | Evaluation on High-sync-op kernels

We now discuss the impact of uClocks on High-sync-op kernels to show that in kernels with many atomic- and clock-operations per second, the performance improves significantly.

7.1.1 | Overall impact of uClocks

Figure 15 shows the raw execution times obtained by using uClocks (includes the proposed extension to X10 clocks, discussed in Section 4.1, and optimizations, discussed in Section 4.2) and Baseline (the original benchmark kernels using the default runtime) on the Intel system. Figure 16 shows the resulting speedup, where Speedup=Time-taken-for-Baseline-version/Time-Taken-for-uClocks-version. The graphs show that we get significant improvement in performance: geomean across varying number of cores between 3.27× - 10.65×; overall geomean = 5.36× on the Intel system.

In general, we find that with increasing number of cores, the improvements more or less increase. In the case of HS and LCR, we see that improvement is quite high for 8-cores. This is because for some reason the Baseline version performs poorly for 8-cores on these benchmarks, because of which our gains for 8-cores look amplified. A similar argument holds for the improvements of KC and BY for 1-core.

The impact of uClocks on the AMD system is similar to that on the Intel system; see Figure 17 for the raw execution times and Figure 18 for speedups. The geomean improvements varied from 5.85× to 12.98× (overall geomean = 11.39×). As shown by evaluations on both the AMD and Intel systems, uClocks lead to improved scaling of the input kernels.

In general, the actual improvement depends on a number of factors such as the input graph, the amount of parallelism present in the input program and the number of advances- and atomic-operations per second.

	1 core		2 core		4 core		8 core		16 core	
	B	U	B	U	B	U	B	U	B	U
KC	29	10	16	7	14	4	19	2	27	2
HS	505	84	401	56	309	37	441	23	466	24
LCR	206	50	198	31	156	21	209	11	221	10
BY	319	194	188	131	89	59	59	37	55	24

FIGURE 15 Raw execution times for uClocks (U) and Baseline (B) on the Intel system in seconds.

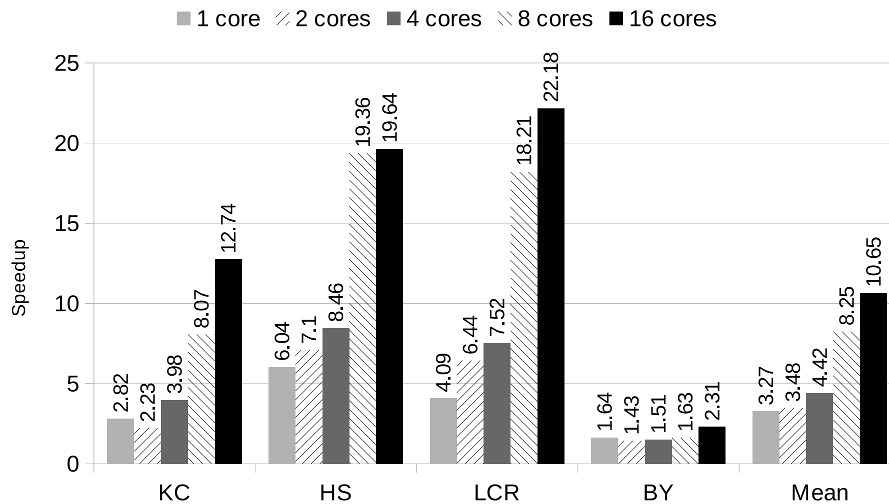


FIGURE 16 Speedups for High-sync-op kernels (Intel system).

	1 core		2 core		4 core		8 core		16 core		32 core		64 core	
	B	U	B	U	B	U	B	U	B	U	B	U	B	U
KC	81	12	44	12	70	6	59	4	56	3	62	3	65	4
HS	1421	107	1476	119	1980	110	1209	83	1036	66	1221	71	1790	82
LCR	548	62	710	62	964	55	577	35	441	28	525	28	764	27
BY	738	252	403	178	215	105	171	63	141	47	126	48	117	45

FIGURE 17 Raw execution times for uClocks (U) and Baseline (B) on the AMD system in seconds.

7.1.2 | Impact of the extension to X10 Clocks and proposed optimizations

We now discuss the impact of the extension to X10 clocks and the optimizations. In Figures 19a and 19b, Column 2 (Speedup from Opt1 only) shows the geomean speedups (across a varying number of cores) achieved due to Opt1 alone, on both the Intel and AMD systems, respectively, for the High-sync-op kernels. Column 3 (Speedup from Opt2 only) shows the incremental speedup obtained by adding Opt2 to this system (X10 clocks + Opt1), or in other words, the speedup which can be attributed to Opt2 alone. Column 4 (Speedup from Language Ext. only) shows the incremental speedup obtained by adding the Language Extension to this system (X10 clocks + Opt1 + Opt2), or in other words, the speedup which can be attributed to the Language Extension alone.

Finally, Column 5 (Overall Speedup) gives a combined speedup of uClocks (X10 clocks + Opt1 + Opt2 + Language Extension) over the original X10 clocks. Hence, as expected, the product of Columns 2, 3 and 4, matches the value in this column. Overall, we see that the proposed extension to the X10 clocks and the optimizations lead to significant improvements, to justify the importance of DP1, DP2, and DP3.

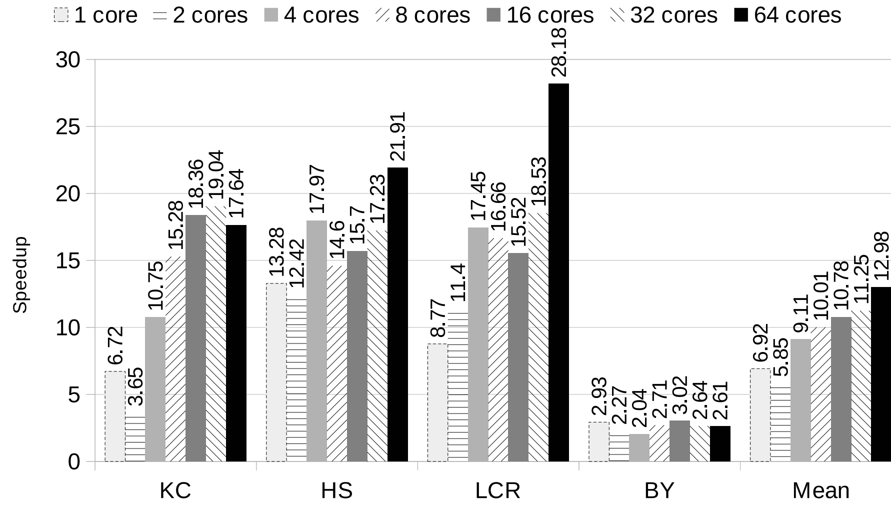


FIGURE 18 Speedups for the High-sync-op kernels (AMD system).

1	2	3	4	5
	Speedup from Opt1 only	Speedup from Opt2 only	Speedup from Language Ext. only	Overall Speedup
KC	2.07	1.27	1.83	4.81
HS	1.00	1.47	7.25	10.66
LCR	0.99	1.41	6.85	9.56
BY	1.52	1.05	1.05	1.68

(a) Intel system.

1	2	3	4	5
	Speedup from Opt1 only	Speedup from Opt2 only	Speedup from Language Ext. only	Overall Speedup
KC	2.88	1.47	2.70	11.43
HS	1.00	1.45	10.99	15.94
LCR	1.00	1.35	11.63	15.70
BY	2.09	1.21	1.02	2.58

(b) AMD system.

FIGURE 19 Speedups due to the proposed optimizations and language extension, for High-sync-op kernels.

	Vs Baseline	Vs uClocks
KC	5.57	1.16
HS	14.89	1.40
LCR	14.29	1.49
BY	1.75	1.04

(a) Intel system.

	Vs Baseline	Vs uClocks
KC	16.49	1.45
HS	42.08	2.65
LCR	38.60	2.46
BY	2.62	1.01

(b) AMD system.

FIGURE 20 Speedup of async-finish kernel versions compared to the Baseline and uClocks versions, for High-sync-op kernels.

7.1.3 | Comparison of code with and without clocks

Even though clocks can be helpful in writing more natural and readable codes[4], a common complaint regarding code using X10 clocks is that it runs significantly slower[22] than its counterparts written without the use of clocks (using only `async-finish`). For example, The 'Vs Baseline' column in Figures 20a and 20b shows the (geomean, across varying number of cores) speedup of the High-sync-op kernels written without clocks to the Baseline versions – up to 14.89× on the Intel system, and 42.08× on the AMD system. The 'Vs uClocks' column shows the corresponding numbers with respect to uClocks– up to 1.49× on the Intel system, and 2.65× on the AMD system. In spite of clocks being helpful in writing more readable codes, the performance gap between codes written with and without clocks has been prohibitively high until now, and our proposed uClocks scheme

	Intel system	AMD system
BF	1.04	1.16
DST	1.06	1.21
MIS	1.02	1.21
MST	1.01	1.16
VC	1.03	1.20
DS	1.02	1.05
DP	1.12	1.19

FIGURE 21 Overall mean speedups for Low-sync-op kernels (mean speedups across varying number of cores).

	Intel system (16 cores)	AMD system (64 cores)
BF	1.20 (1.17, 1.24)	1.15 (1.13,1.18)
DST	1.08 (1.07,1.09)	1.24 (1.22,1.26)
MIS	1.10 (1.09, 1.11)	1.36 (1.35,1.38)
MST	1.05 (1.04, 1.06)	1.16 (1.15,1.17)
VC	1.09 (1.08,1.10)	1.29 (1.20,1.38)
DS	1.15 (1.14, 1.16)	1.18 (1.18,1.19)
DP	1.21 (1.20,1.23)	1.19 (1.18,1.19)

FIGURE 22 Speedup ratios and their corresponding confidence intervals (reported in brackets) for Low-sync-op kernels on the 16-core Intel system and 64-core AMD system.

significantly reduces this gap, thereby making the use of clocks more attractive. However, even with our proposed optimization, the code using clocks are still slower (geomean 35% slower). It remains an interesting future work to bridge this gap.

7.2 | Evaluation on Low-sync-op kernels

We now discuss the impact of uClocks on Low-sync-op kernels to show that in kernels with not many atomic- and clock-operations per second, the performance does not deteriorate.

7.2.1 | Overall impact of uClocks

We found that for the Low-sync-op kernels, the uClocks scheme does not lead to much difference in performance compared to the Baseline. Figure 21 summarizes the geomean speedups achieved by the uClocks scheme, across varying number of cores, for both the Intel and AMD systems. Note that the slightly higher geomean gains on the AMD system, as compared to the Intel system, are due to the underlying differences in the architecture and such improved gains have been observed across all the kernels (both High-sync-op and Low-sync-op). Since the speedups in Figure 21 are small (close to 1×), we also report the confidence intervals for the speedup ratio (as defined by Kalibera and Jones[23]) for the 16-core Intel system and 64-core AMD system in Figure 22. The narrow width of the confidence intervals shows that the execution time is fairly stable across different runs.

7.2.2 | Comparison of code with and without clocks

The 'Vs Baseline' column in Figures. 23a and 23b shows that in the Low-sync-op kernels, the deterioration due to clocks is less. The 'Vs uClocks' column shows that in most cases, the uClocks scheme removes this minor deterioration and even improves the performance. This makes uClocks an attractive option even for Low-sync-op programs. MST shows an interesting case, where both the Baseline and uClocks versions perform better than the async-finish version. This is because the latter includes a series of parallel-for-loops leading to significant task-creation and termination overheads, which is avoided in the former because of

	Vs Baseline	Vs uClocks
BF	1.14	1.10
DST	1.02	0.97
MIS	0.96	0.94
MST	0.87	0.86
VC	1.04	1.01
DS	0.94	0.91
DP	1.04	0.92

(a) Intel system.

	Vs Baseline	Vs uClocks
BF	1.14	0.99
DST	1.08	0.89
MIS	1.10	0.91
MST	0.89	0.76
VC	1.00	0.83
DS	0.97	0.93
DP	1.13	0.95

(b) AMD system.

FIGURE 23 Mean speedup of async-finish kernel versions compared to the Baseline and uClocks versions, for Low-sync-op kernels (mean speedups across varying number of cores).

	Vs Baseline	Vs uClocks
BF	1.42 (1.37, 1.47)	1.18 (1.16, 1.20)
DST	1.12 (1.11, 1.13)	1.04 (1.04, 1.05)
MIS	1.04 (1.03, 1.05)	0.95 (0.94, 0.96)
MST	0.87 (0.86, 0.89)	0.83 (0.82, 0.85)
VC	1.22 (1.20, 1.23)	1.12 (1.10, 1.13)
DS	1.10 (1.09, 1.12)	0.96 (0.94, 0.97)
DP	1.23 (1.22, 1.25)	1.01 (1.01, 1.02)

(a) Intel system (16 cores)

	Vs Baseline	Vs uClocks
BF	1.11 (1.09, 1.13)	0.96 (0.94, 0.98)
DST	1.16 (1.14, 1.18)	0.93 (0.93, 0.94)
MIS	1.28 (1.26, 1.30)	0.94 (0.93,0.95)
MST	0.87 (0.87, 0.88)	0.75 (0.75, 0.76)
VC	1.08 (1.01, 1.15)	0.84 (0.81, 0.86)
DS	1.07 (1.06, 1.08)	0.91 (0.90, 0.91)
DP	1.19 (1.18, 1.19)	1.00 (1.00,1.00)

(b) AMD system (64 cores)

FIGURE 24 Speedup ratios and their corresponding confidence intervals (reported in brackets) for the async-finish kernel versions compared to the Baseline and uClocks versions, for Low-sync-op kernels on the 16-core Intel machine and 64-core AMD machine.

the use of clocks. Since the speedups in Figure 23 are small (close to 1×), we also report the confidence intervals for the speedup ratio (as defined by Kalibera and Jones[23]) of the async-finish kernel versions compared to the Baseline and uClocks versions, for two of the highest configurations (the 16-core Intel system and 64-core AMD system in Figure 24).

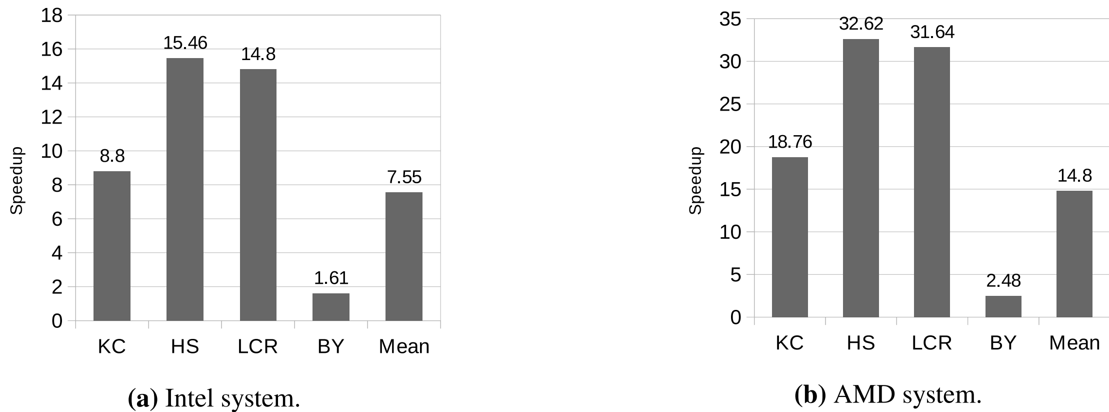
7.3 | Evaluation of the kernels on C++ backend

We have also evaluated the impact of our optimizations on the C++ backend of the X10 compiler. For brevity, in Figure 25, we only show the geomean speedups across the number of cores for the High-sync-op kernels, on both the Intel and AMD systems. Overall we find that the speedups are equally encouraging for the C++ backend as well.

Summary of Evaluation. The performance evaluation (across both the C++ and Java backends) shows that the uClocks scheme provides substantial benefits for kernels with a significant number of synchronization operations per second, and minor benefits (but no degradation) for benchmarks with fewer synchronization operations per second. On the whole, this makes it a desirable implementation for both kinds of workloads. The large performance improvement empirically justifies the importance of the properties mentioned in Section 3 for a Runtime implementation (for languages with fine-grained synchronization). We also believe that the set of desirable properties and the proposed optimizations are also meaningful in the context of other PGAS task-parallel languages (like HJ and Chapel) that allow lock-step synchronization among tasks.

8 | RELATED WORK

Efficient Runtime design for languages that support fine-grain synchronization. There have been many prior efforts [24,25,26] on improving the runtime for task-parallel languages that support synchronization. There has also been prior work on making



(a) Intel system.

(b) AMD system.

FIGURE 25 Speedups for uClocks over X10 Clocks with the C++ backend, for the High-sync-op kernels.

the execution of `async-finish` constructs in X10 efficient. Guo et al.[11] present two variations of the work-stealing scheduler (help-first and work-first) for X10 (with `async-finish` constructs only), which improve on the performance of the work-sharing scheduler. Both Kumar et al.[27] and Tardieu et al.[28] extend the work-stealing scheduler in X10. Kumar et al. improve the performance by significantly reducing its sequential overheads. Tardieu et al. extend the idea of work-stealing to other constructs like `when`, and clocks, by allowing suspension of tasks at the synchronization point (implemented using continuations – can be expensive). We also believe that the abstraction of Fibers from the project Loom[29], which enable the creation of lightweight user-mode threads, can be used to efficiently implement the runtime for languages that support fine-grain synchronization. Imam and Sarkar[8] present a mixed (compile-time+runtime) approach for HJ[2] that uses One-Shot-Delimited-Continuations to store the context at the synchronization point (and switch to other pending tasks), and Event-Driven-Controls to efficiently resume suspended tasks. Shirako and Sarkar[9] use a hierarchical phaser organization to reduce lock contention and allow scalable synchronization among tasks. In contrast, based on a set of desirable properties that are applicable across multiple task-parallel languages, we present an extension to X10 clocks and design new optimizations to efficiently implement clocks in the work-stealing scheduler without requiring switching of tasks (suspension) or storing of continuation contexts.

Efficient compilation of clocks and related operations. Vasudevan et al.[30] present a compile time technique to identify clock-objects that (i) do not throw ClockUseException[10]. (ii) do not call `resume` explicitly, and (iii) are only used at the current X10 place[10]. Such clock-objects are replaced with instantiations of specialized clock classes that take advantage of the above properties. Nandivada et al.[31] present techniques to reduce the overheads of X10 clock (and HJ phaser) operations by chunking parallel loops with synchronization operations. Feautrier et al.[22] propose a technique to transform code written using `clocks-async-finish` abstractions to code that does not use `clocks`. However, their scheme works for static-control-programs; hence, it covers only for a restricted class of parallel-loops and unconditional `advance` operations. In contrast, we propose a languages extension and an improved runtime that supports arbitrary X10 programs. Further, the existing compile time optimizations can be used along with our proposed techniques in an orthogonal manner.

Barriers in task-parallel languages. Barriers in Chapel[3], clocks in X10 and phasers[32] in HJ allow fine grain synchronization among tasks. Such barrier synchronization constructs have been formalized by prior researchers[33,34,35,36], for languages like X10 and HJ. Languages like OpenMP[12] support synchronization of worker-threads (but not tasks) using barriers. Since programming thread-level barriers in a task-parallel language is tedious, Aloor and Nandivada[18,19] introduce an extension to OpenMP called UW-OpenMP which allows the programmer to specify barriers among tasks, and then use a source-to-source translation to convert UW-OpenMP code to equivalent OpenMP code that does not require task-barriers. Shirako et al.[20] propose the idea of introducing *Phasers* (similar to that in HJ) to enable fine grain synchronization of OpenMP threads. In this paper, we extend X10 clocks to additionally admit lazy resume and advance operations, which leads to significant performance gains.

9 | CONCLUSION

While clocks provide a convenient, high-level abstraction to specify fine-grained synchronization in task-parallel programs, its associated implementation overheads compared to the `async-finish` (fork-join) counter-parts can be prohibitively high. To

bridge the gap, we propose three properties that an efficient runtime system for languages that support lock-step synchronization should try to satisfy. Based on these desirable properties, we present a scheme called uClocks to improve the efficiency of X10 clocks; uClocks consists of an extension to X10 clocks and two optimizations to the X10 runtime. We prove that uClocks satisfies the desirable properties. We have evaluated uClocks on two different hardware setups and found that for benchmarks that use a large number of synchronization operations, uClocks leads to significant speedups (geomean 5.36× on a 16 core Intel system, and 11.39× on a 64 core AMD system). Though we have implemented uClocks in the context of X10, we believe that it can also be extended to other task-parallel programming languages like HJ and Chapel.

10 | REFERENCES

References

1. Charles Philippe, Grothoff Christian, Saraswat Vijay, et al. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In: OOPSLA '05:519–538ACM; 2005; New York, NY, USA.
2. Cavé Vincent, Zhao Jisheng, Shirako Jun, Sarkar Vivek. Habanero-Java: The New Adventures of Old X10. In: PPPJ '11:51–61ACM; 2011; New York, NY, USA.
3. The Chapel Parallel Programming Language https://chapel-lang.org.
4. Yuki Tomofumi. Revisiting Loop Transformations with x10 Clocks. In: X10 2015:1–6ACM; 2015; New York, NY, USA.
5. Shirako Jun, Zhao Jisheng M., Nandivada V. Krishna, Sarkar Vivek N.. Chunking Parallel Loops in the Presence of Synchronization. In: ICS '09:181–192ACM; 2009; New York, NY, USA.
6. X10 Version 2.6.1 Runtime https://github.com/x10-lang/x10.
7. Gupta Suyash, Nandivada V. Krishna. IMSuite: A benchmark suite for simulating distributed algorithms. *Journal of Parallel and Distributed Computing.* 2015;75(0):1 - 19.
8. Imam Shams, Sarkar Vivek. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In: :618–643Springer-Verlag New York, Inc.; 2014; New York, NY, USA.
9. Shirako J., Sarkar V.. Hierarchical phasers for scalable synchronization and reductions in dynamic parallelism. In: IPDPS'2010:1-12; 2010.
10. Saraswat Vijay, Bloom Bard, Peshansky Igor, Tardieu Olivier, Grove David. X10 Language Specification Version 2.6.1 http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf.
11. Guo Yi, Barik Rajkishore, Raman Raghavan, Sarkar Vivek. Work-first and Help-first Scheduling Policies for Async-finish Task Parallelism. In: IPDPS '09:1–12IEEE Computer Society; 2009; Washington, DC, USA.
12. OpenMP Specification http://www.openmp.org/specifications/.
13. Gosling James, Joy Bill, Steele Guy, Bracha Gilad, Buckley Alex. Java version 1.8 Specification https://docs.oracle.com/javase/specs/jls/se8/html/index.html.
14. Unified Parallel C https://upc-lang.org.
15. Krishnamurthy A., Culler D. E., Dusseau A., et al. Parallel Programming in Split-C. In: Supercomputing '93:262–273ACM; 1993; New York, NY, USA.
16. Numrich Robert W., Reid John. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum.* 1998;17(2):1–31.
17. Yelick Kathy, Semenzato Luigi, Pike Geoff, et al. Titanium: A High-Performance Java Dialect. In: :10–11; 1998.
18. Aloor Raghesh, Nandivada V. Krishna. Unique Worker Model for OpenMP. In: ICS '15:47–56ACM; 2015; New York, NY, USA.

19. Aloor Raghesh, Nandivada V. Krishna. Efficiency and Expressiveness in UW-OpenMP. In: CC 2019:182–192ACM; 2019; New York, NY, USA.

20. Shirako Jun, Sharma Kamal, Sarkar Vivek. Unifying Barrier and Point-to-point Synchronization in OpenMP with Phasers. In: IWOMP'11:122–137Springer-Verlag; 2011; Berlin, Heidelberg.

21. Georges Andy, Buytaert Dries, Eeckhout Lieven. Statistically rigorous Java performance evaluation. In: OOPSLA '07:57–76ACM; 2007; New York, NY, USA.

22. Feautrier Paul, Violard Eric, Ketterlin Alain. Improving X10 Program Performances by Clock Removal. In: CC'14; 2014; Grenoble, France.

23. Kalibera Tomas, Jones Richard. Rigorous Benchmarking in Reasonable Time. In: ISMM '13:63–74ACM; 2013; New York, NY, USA.

24. Olivier Stephen L, Porterfield Allan K, Wheeler Kyle B, Spiegel Michael, Prins Jan F. OpenMP task scheduling strategies for multicore NUMA systems. *The International Journal of High Performance Computing Applications.* 2012;26(2):110-124.

25. Blumofe Robert D., Joerg Christopher F., Kuszmaul Bradley C., Leiserson Charles E., Randall Keith H., Zhou Yuli. Cilk: An Efficient Multithreaded Runtime System. In: PPOPP '95:207–216ACM; 1995; New York, NY, USA.

26. Yang Jixiang, He Qingbi. Scheduling Parallel Computations by Work Stealing: A Survey. *International Journal of Parallel Programming.* 2018;46(2):173–197.

27. Kumar Vivek, Frampton Daniel, Blackburn Stephen M., Grove David, Tardieu Olivier. Work-stealing Without the Baggage. In: OOPSLA '12:297–314ACM; 2012; New York, NY, USA.

28. Tardieu Olivier, Wang Haichuan, Lin Haibo. A Work-stealing Scheduler for X10's Task Parallelism with Suspension. In: PPoPP '12:267–276ACM; 2012; New York, NY, USA.

29. Pressler Ron, Bateman Alan. *Project Loom - Fibers, Continuations and Tail-Calls for the JVM.* 2018.

30. Vasudevan Nalini, Tardieu Olivier, Dolby Julian, Edwards Stephen A. Compile-Time Analysis and Specialization of Clocks in Concurrent Programs. In: CC '09:48–62Springer-Verlag; 2009; Berlin, Heidelberg.

31. Nandivada V. K., Shirako J., Zhao J., Sarkar V.. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.*. 2013;35(1):3:1–3:48.

32. Shirako Jun, Peixotto David M., Sarkar Vivek, Scherer William N.. Phasers: A Unified Deadlock-free Construct for Collective and Point-to-point Synchronization. In: ICS '08:277–288ACM; 2008; New York, NY, USA.

33. Martins Francisco, Vasconcelos Vasco T., Cogumbreiro Tiago. Types for X10 Clocks. In: EPTCS, vol. 69: :111–129; 2010.

34. Cogumbreiro Tiago, Martins Francisco, Thudichum Vasconcelos Vasco. Coordinating Phased Activities while Maintaining Progress. In: De Nicola Rocco, Julien Christine, eds. *Coordination Models and Languages*, :31–44Springer Berlin Heidelberg; 2013; Berlin, Heidelberg.

35. Cogumbreiro Tiago, Shirako Jun, Sarkar Vivek. Formalization of Habanero phasers using Coq. *Journal of Logical and Algebraic Methods in Programming.* 2017;90:50–60.

36. Cogumbreiro Tiago, Shirako Jun, Sarkar Vivek. Formalization of phase ordering. In: EPTCS, vol. 211: :13–24; 2016. Proof scripts.

APPENDIX

A | SETUP FOR RUNNING EXPERIMENTS

A.1 Abstract

This appendix describes how to replicate the experiments to evaluate the performance of uClocks as compared to the original X10 Clocks implementation. It details the software and hardware requirements, installation instruction, steps to configure and run the experiment, and the expected results.

A.2 Description

A.2.1 Overview

- **Programming language**: X10.
- **Compilation**: X10-2.6.1 compiler and runtime
- **Data set**: IMSuite Benchmarks (https://www.cse.iitm.ac.in/~krishna/imsuite/)
- **Hardware**: System with atleast 16 cores
- **Output**: Execution time of all Kernels on a) uClocks b) original X10 Clocks
- **Experiment workflow**: (i) Build original X10 compiler+runtime, (ii) Build X10 compiler+runtime extended with uClocks, (iii) Run performance measurements on original X10 compiler+runtime, (iv) Run performance measurements on X10 compiler+runtime extended with uClocks
- **Experiment customization**: Number of cores to run on, Number of runs for each benchmark
- **Publicly available?**: Yes

A.2.2 How software can be obtained

All required software can be downloaded from https://github.com/akshayutture/uClocks

A.2.3 Hardware dependencies

An active internet connection is required to download the necessary jar files and a multi-core system with at least 16 cores.

A.2.4 Software dependencies

The dependencies are given on the X10 website (http://x10-lang.org/x10-development/building-x10-from-source.html)

A.2.5 Datasets

We use the IMSuite benchmarks for simulating distributed algorithms, which are available at https://www.cse.iitm.ac.in/~krishna/imsuite/. We use all the eleven Iterative FAC (Finish-Async-Clocks) Kernels for X10-2.5.0.

The input size used is 512 for all kernels except the byzantine and dominatingSet kernels which use an input size of 128.

A.3 Installation

Run the following two commands to quickly build the two versions of X10 (the first being the original X10 language+runtime, and the second being the X10 language+runtime extended with uClocks)

```
$ cd uClocks
$ sh compile-X10Clocks.sh
$ sh compile-uClocks.sh
```

A.4 Experiment workflow

A.4.1 Run benchmarks with original X10 clocks

Run the following command to get the readings of the IMSuite benchmarks (IMSuite-IterativeKernels) on the uClocks extended runtime, in the file 'originaloutput.txt'.

```
$ python run-X10Clocks-on-IMSuite-kernels.py [numberOfCores] [numberOfRunsPerBenchmark] > originaloutput.txt
```

In case the benchmarks run into *Out of Memory* or *Out of Heap space errors*, run the following command instead, which uses smaller sized benchmarks, but will not give the same quality of results (recommended that you run on a more powerful system, instead of using this).

```
$ python run-X10Clocks-on-IMSuite-kernels_small-Inputs.py [numberOfCores] [numberOfRunsPerBenchmark] > originaloutput.txt
```

A.4.2 Run benchmarks with uClocks extended runtime

Run the following command to get the readings of the IMSuite benchmarks (IMSuite-IterativeKernels) on the uClocks extended runtime, in the file 'uClocksoutput.txt'.

```
$ python run-uClocks-on-IMSuite-kernels.py [numberOfCores] [numberOfRunsPerBenchmark] > uClocksoutput.txt
```

In case the benchmarks run into *Out of Memory* or *Out of Heap space errors*, run the following command instead, which uses smaller sized benchmarks, but will not give the same quality of results (recommended that you run on a more powerful system, instead of using this).

```
$ python run-uClocks-on-IMSuite-kernels_small-Inputs.py [numberOfCores] [numberOfRunsPerBenchmark] > uClocksoutput.txt
```

A.5 Evaluation and expected result

The ratios of the execution time obtained from the two runs (with original X10 clocks and uClocks) in the output file, can be plotted graphically, and the characteristics of the overall speedups may match (depending on the hardware used) those obtained in Figure 16 or Figure 18. uClocks perform significantly better for the High-sync-op kernels described in Section 7.1.1.

A.6 Notes

For any further issues or queries in the installation, kindly refer the X10 website
http://x10-lang.org/x10-development/building-x10-from-source.html, the SourceForge page
https://sourceforge.net/p/x10/mailman/x10-users/ or the Stack overflow page
https://stackoverflow.com/questions/tagged/x10-language.