Deadlock-Free Scheduling of X10 Computations with Bounded Resources

Shivali Agarwal [*]
shivali@tcs.tifr.res.in

Rajkishore Barik [†]
rajbarik@in.ibm.com

Dan Bonachea [‡§]
bonachea@cs.berkeley.edu

Vivek Sarkar [¶]
vsarkar@us.ibm.com

Rudrapatna K Shyamasundar [†]
rshyamas@in.ibm.com

Katherine Yelick [‡§]
yelick@cs.berkeley.edu

[*] Tata Institute of Fundamental Research, Mumbai, India
[†] IBM India Research Lab, New Delhi, India
[‡] Computer Science Division, University of California at Berkeley, California, USA
[§] Lawrence Berkeley National Laboratory, California, USA
[¶] IBM T.J. Watson Research Center, New York, USA

ABSTRACT

In this paper, we address the problem of guaranteeing the absence of *physical deadlock* in the execution of a parallel program using the `async`, `finish`, `atomic` and `place` constructs from the X10 language. First, we extend previous work-stealing memory bound results for *fully strict* multithreaded computations to *terminally strict* multithreaded computations in which one activity may wait for completion of a descendant activity (as in X10's `async` and `finish` constructs), not just an immediate child (as in Cilk's `spawn` and `sync` constructs). This result establishes physical deadlock freedom for SMP deployments. Second, we introduce a new class of X10 deployments for clusters, which builds on an underlying Active Message network and the new concept of Doppelgänger mode execution of X10 activities. Third, we use this new class of deployments to establish physical deadlock freedom for deployments on clusters of uniprocessors.

Together these results give the user the ability to execute a rich set of programs written with `async`, `finish`, `atomic` and `place` constructs without worrying about the possibility of physical deadlock due to computation, memory and communication resources. A major open topic for future work is to extend these results to deployments on clusters of SMPs.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming, parallel programming*

General Terms

Algorithms, Languages, Theory

Keywords

X10, Deadlock-free scheduling, Active Messages

1. INTRODUCTION

Driven by the ever-increasing levels of hardware parallelism and the need to mask memory and network latencies, the DARPA HPCS languages all combine dynamic parallelism with locality control. This raises the question of how to map dynamic parallelism onto a fixed set of processor and memory resources while obeying the locality properties of the program. X10 is a novel object-oriented programming language designed for high-performance, high-productivity programming of parallel computers [6] as part of the DARPA HPCS program. X10 differentiates itself from multithreaded programming models such as Cilk [7] through its explicit reification of locality in the form of *places*, which serve as virtual locations for distributed activities and data. X10 also differentiates itself from Partitioned Global Address Space (PGAS) languages [10, 14, 16] and other Single Program Multiple Data (SPMD) programming models through its integration of loop parallelism, task parallelism, and asynchronous communication in a single unified execution model.

X10 creates parallel activities using an `async` statement, forces their completion with `finish`, supports mutual exclusion with `atomic`, and provides a generalized form of barrier synchronization called *clocks*. The async statement generalizes other dynamic threading models by allowing the programmer to specify a place at which the computation should be performed. In past work [15], it was shown that X10 programs written with `async`, `finish`, `atomic` and `clocked` are guaranteed to be free of *logical deadlock i.e.,* they are guaranteed to never deadlock when executed with *unbounded* computation, memory, and communication resources.

This paper studies the problem of deadlock-free scheduling of X10 programs on physical nodes with *bounded* computation, memory, and communication resources. For simplicity, we focus on a core subset of the X10 language consisting of `async`, `atomic` and `finish`. In general, physical nodes have bounded *computation resources* (which limit the maximum number of activities that can execute simultaneously), bounded *memory resources* (which limit the number of live activities that can be suspended), and bounded *communication resources* (which limit the maximum number of entries in a node's input/output communication queue).

In Section 2, we restrict our attention to *single-place* X10 computations which correspond to SMP deployments for which communication resource bounds do not apply. We address the use of bounded computation and memory resources for this case through the use of *work stealing* and show that for an SMP with P processors, X10 programs written with **async** and **finish** will use space no greater than $S_1 * P$, where S_1 is the space used by a single processor execution. This generalizes previous work-stealing results for the Cilk language [3] to admit more general synchronization patterns in which one activity may wait for completion of a descendant activity, not just an immediate child as in Cilk.

In Section 3, we introduce the notion of X10 *deployments* which formalize the mapping of *virtual* places to *physical* nodes by adding bounded communication resources. Unfortunately, a simple one-to-one mapping from places to nodes for data and activities may lead to *physical deadlock* due to the bounds on stalled activities and communication queues. To address this problem, we propose a communication framework built on *Active Messages* and an extension of X10 deployments with a *Doppelgänger* capability (Section 3.3), which enables an X10 activity to run on a node other than the node specified by the programmer. Doppelgänger mode enables us to create deployments for clusters of uniprocessors that are guaranteed to never exhibit physical deadlock by exploiting deadlock-free communication protocols such as Active Messages.

In Section 4, we discuss a set of scheduling algorithms for multi-place X10 computations using **async**, **finish** and **atomic** that are proved to a) never exhibit physical deadlock, and b) stay within the memory bounds of work stealing algorithms. The scheduling algorithms include heuristics that attempt to obey the place directives provided by the X10 programmer as far as possible, and only ignore them by entering Doppelgänger mode when necessary to satisfy a) and b).

Finally, Sections 5 and Section 6 summarize related work and the conclusions, and Appendix A includes proofs and additional technical details.

2. BOUNDED-MEMORY SCHEDULING OF X10 PROGRAMS ON SMP'S

In this section, we show how a *single place* deployment of an X10 program on a SMP with P physical processors can be executed in a deadlock-free manner, while satisfying per-processor memory bounds. Recall that inter-processor communication resources are not an issue in this single place scenario.

2.1 X10 computations

As in Cilk [3], an X10 computation can be represented as a *dag* in which each node corresponds to a dynamic execution instance of an X10 instruction/statement, and each edge defines a precedence constraint between two nodes. The first instruction of the main activity [1] serves as the *root* node of the dag (with no predecessors). Any instruction which spawns a new activity will create a child node in the dag with an edge from the spawn instruction to the first instruction of that child activity. These edges are called *spawn* edges. In addition, execution instances of instructions from the same activity are sequenced in the dag by the use of *continue* edges.

[1]The terms "activity" and "thread" are used interchangeably in this paper.

X10 activities may wait on descendant activities by executing a **finish** statement. We model these dependencies by introducing **start fin** and **end fin** nodes in the dag for each instance of a **finish** construct and then create *dependence* edges from the last instruction of the spawned activities within the scope of **finish** to the corresponding **end fin** instruction. Figure 1 shows an example X10 code fragment and its computation dag.

```
l1: S0;
l2: finish async {
l3:   S1;
l4:   async {
l5:     S2;}
l6:   S3;}
l7: S4;
```

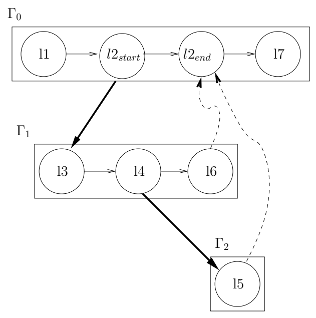


Figure 1: X10 computation dag

While there are many similarities between an X10 computation dag and a Cilk computation dag, there are also some interesting differences. For example, the direct dependence edge from activity Γ_2 to activity Γ_0 in Figure 1 is not allowed in Cilk because Γ_0 is not Γ_2's parent in the dag. The only way to establish such a dependence in Cilk is via Γ_1. In X10 it is also possible for a descendant activity (e.g., Γ_2) to continue executing even if its ancestor activity (e.g., Γ_1) has terminated. This degree of asynchrony can be useful in parallel divide-and-conquer algorithms so as to permit sub-computations at different levels of the divide-and-conquer tree to execute in parallel without forcing synchronization at the parent-child level.

We now show how the Work-Stealing Algorithm from [3] for *fully strict* multithreaded computations can be extended to *terminally strict* multithreaded computations in which one activity may wait for completion of a descendant activity.

2.2 Work-Stealing Algorithm: Overview

In this section, we summarize key definitions and results from [3] that provide relevant background for our paper.

- An *execution schedule*, denoted by $\mathcal{X}$, for a multithreaded computation determines which processors of a parallel computer execute which instructions at each step.
- The space required by a P-processor execution schedule, $\mathcal{X}$, is denoted by $S(\mathcal{X})$.
- The space required by a 1-processor execution schedule is denoted by S_1.

- S_{max} denotes the size in bytes of the largest activation frame in the computation.
- The *stack depth* of an activity is defined to be the sum of the sizes of the activation frames of all its ancestors, including itself.
- The *stack depth* S, expressed in bytes, of a multithreaded computation is the maximum stack depth of any activity.
- The *busy-leaves property*: At every time step, every living thread that has no living descendant has a processor working on it.
- In a *strict* multithreaded computation, every dependency edge goes from a thread to one of its ancestor threads.
- In a *fully strict* multithreaded computation, every dependency edge goes from a thread to its parent thread.

Every strict multithreaded computation is a depth first computation and obviously, fully strict computations are a proper subset of strict computations.

THEOREM 2.1. *For any depth-first multithreaded computation with stack depth S, we have $S_1 = S$.*
Refer to [3] for proof.

LEMMA 2.2. *For any multithreaded computation with stack depth S_1, any P-processor execution schedule, $\mathcal{X}$, that maintains the busy-leaves property uses space bounded by $S(\mathcal{X}) \leq S_1 P$.*
Refer to [3] for proof.

THEOREM 2.3. *For any fully strict multithreaded computation with stack depth S_1, the Work-Stealing Algorithm run on a computer with P processors uses at most $S_1 P$ space.*
Refer to [3] for proof.

2.3 Extending Work-Stealing for X10

The difference between Cilk and X10 computations is that unlike Cilk, X10 does not fall into the class of *fully strict* multithreaded computations and, therefore, we cannot directly apply the result from Theorem 2.3. However, X10 computations fall in the class of *strict* multithreaded computations because dependence edges can only occur from descendants to ancestors. Further, X10 computations that use `async`, `finish` and `atomic` constructs form a proper subset of strict computations that we refer to as *terminally strict* multithreaded computations. We use "*terminally*" to emphasize the fact that dependencies from descendants to ancestors only occur on *termination* of descendant activities, as determined by X10's `finish` construct.

Specifically, a terminally strict multithreaded computation must satisfy the following properties, where $\Gamma_0, \Gamma_1, \Gamma_2$, $\ldots$ Γ_n represents a chain of activities such that Γ_{j+1} is a child of Γ_j for $j = 1 \ldots n$:

1. In case of dependencies on descendants of a child, it is always the case that a dependency on the child also exists.
2. All the descendants created within the immediate scope of `finish` will have a dependence edge going to the `end fin` node of that `finish`.
3. There can be at most one outgoing dependence edge from a node. Note that in absence of dependence edge between two activities, they can complete in any order.
4. When an activity, say Γ_0, with an outgoing dependence edge completes, the dependency on Γ_0 is resolved. An activity with incoming dependence edges can progress only when all the edges get resolved. For example, in Figure 1, Γ_0 can progress past its `end fin` instruction only when Γ_1 and Γ_2 complete. Note that Γ_1 and Γ_2 can complete in any order.
5. A `finish` block completes only when all its descendant activities complete. In case of nested `finish` blocks, this will imply transitive dependency.

The Work-Stealing algorithm for scheduling X10 computations provably uses bounded space and can be shown not to have any cyclic dependencies thus guaranteeing that computation will not exhibit deadlock due to resource constraints. Furthermore, the overall space bound will be demonstrated to scale no greater than linearly with increasing processor count, which is generally a precondition for practical implementation.

Each processor maintains a *ready deque* data structure of threads. The ready deque has two ends: a *top* and a *bottom*. Threads can be inserted in the bottom and removed from either end. A processor treats its ready deque like a call stack, pushing and popping from the bottom. Threads that are migrated to other processors are removed from the top.

In general, a processor obtains work by removing the thread at the bottom of its ready deque. It starts working on the thread, call it Γ_a, and continues executing Γ_a's instructions until Γ_a spawns, blocks, dies, or enables a blocked thread, in which case, it performs according to the following rules:

1. **Async**: If the thread Γ_a spawns an async child Γ_b, then Γ_a is placed on the bottom of the ready deque, and the processor commences work on Γ_b.
2. **Blocks**: If the thread Γ_a blocks due to a `finish` construct, its processor begins work stealing. A thread can block if there exists some live descendants from which it has incoming dependence edges. The only possible reason that this thread got separated from its descendants is that it was stolen.
3. **Terminates**: If the thread Γ_a terminates, it first checks if it enables thread Γ_b *i.e.,* if Γ_a is the last dependency for Γ_b's `finish` operator. If so, Γ_b is inserted as the bottommost thread in deque of Γ_a's processor. Next, Γ_a's processor checks its ready deque. If the deque contains any threads, then the processor removes and begins work on the bottommost thread. If the ready deque is empty, the processor begins work stealing: it steals the topmost thread from the ready deque of a randomly chosen processor and begins work on it.

Having outlined the rules of work stealing on a thread (activity) in X10 we now arrive at the space bound in the following manner. We apply the rules of work stealing for fully-strict computations in [3] to terminally-strict X10 computations without much modification, and obtain an analogous space bound.

LEMMA 2.4. *In the execution of any terminally-strict (X10) computation by the Work-Stealing Algorithm, let us consider any processor p and at the beginning of a given time step let there be k number of activities in p's ready deque. Let Γ_0 be the activity that p is working on. Let Γ_1, $\Gamma_2, \ldots \Gamma_k$ denote the k activities ordered from bottom to top, that is, Γ_1 is*

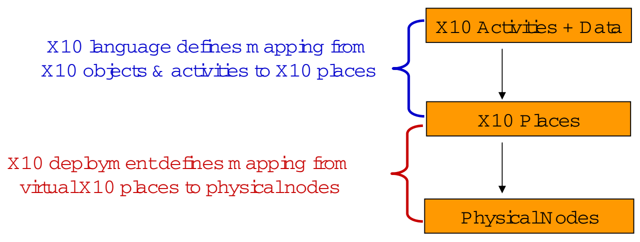

X10 language defines mapping from
X10 objects & activities to X10 places
X10 deployment defines mapping from
virtual X10 places to physical nodes
X10 Activities + Data
X10 Places
Physical Nodes

Figure 2: X10 deployments

the bottommost and Γ_k *is the topmost. If* $k > 0$*, then the activities in* p*'s ready deque satisfy the following properties:*

1. *For* $i = 1, 2, \ldots, k$*, activity* Γ_{i-1} *is the child of* Γ_i*.*
2. *For* $i = 1, 2, \ldots, k$*, activity* Γ_i *has not been worked upon since it spawned* Γ_{i-1}*.*

See section A.1 in appendix for proof.

LEMMA 2.5. *The Work-Stealing Algorithm for terminally-strict* X10 *computations preserves the busy-leaves property.* See section A.1 in appendix for proof.

THEOREM 2.6. *For an* X10 *computation, which is a terminally-strict multithreaded computation, with stack depth* S_1*, the Work-Stealing Algorithm run on a computer with* P *physical processors uses at most* S_1 *bytes for the deque per processor. That is, it uses total space of* $S_1 * P$ *bytes.*

PROOF. Follows from Lemma 2.5 and 2.2. □

This theorem states clearly that using bounded space we can handle any terminally-strict computation, provided that computation requires finite stack depth. This result implies that given $S_1 * P$ space in our system, we can guarantee the progress of the X10 computation using the Work-Stealing Algorithm and will not observe any deadlocks due to insufficient activity memory resources.

3. X10 DEPLOYMENTS, ACTIVE MESSAGES, DOPPELGANGER MODE

3.1 X10 Deployments

As described in [6], a key feature of the X10 programming model is the concept of a *place* which establishes a binding between a set of *activities* and a set of *shared mutable* locations in a partitioned global shared address space. Places are a *virtual* concept. The mapping of places to physical processors and memories is called a *deployment*, as shown in Figure 2. A deployment for an X10 program execution is specified separately from the X10 program – it may be specified by the programmer or application end-user through the run-time environment, or it may be selected automatically by an X10 implementation. Though objects and activities do not migrate across places in an X10 program, an X10 implementation is free to adaptively optimize a deployment at run-time by migrating places across physical locations as it sees fit.

In this section, we focus our attention on the case where the deployment target is a *cluster of uniprocessors* interconnected with a *communication network*. Even though this case represents a restricted subset of potential X10 deployment targets, it serves as a good foundation for establishing deadlock avoidance in the presence of bounded resources. Figure 3 outlines a *standard cluster deployment* where there

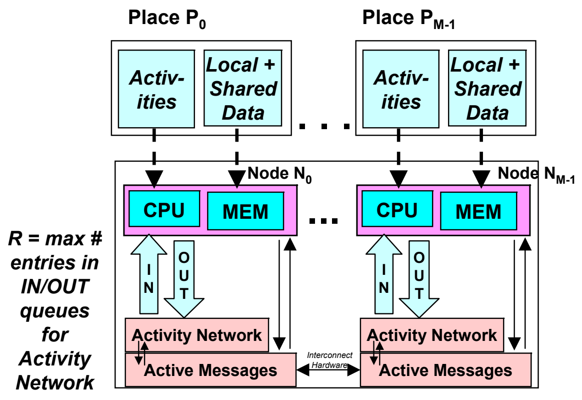


Figure 3: Standard Cluster Deployment; One-to-one mapping of activities and data from places to nodes

is a one-to-one mapping between X10 places $P_0 \ldots P_{M-1}$ and physical nodes $N_0 \ldots N_{M-1}$ — each activity from place P_i in the X10 program is assigned to run on node N_i's processor in the cluster, and each object or array subregion from place P_i is allocated in node N_i's memory.

As indicated in Figure 3, we partition the communication software stack into an *Activity Network* and an *Active Message Network*. The Active Message Network is used to implement the Activity Network, and also to support implicit place-remote accesses in the partitioned global address space. Active Messages (AM)[2] is a low-level lightweight RPC mechanism that supports unordered, reliable delivery of matched request/reply messages. The AM communication pattern is strictly Request/Reply — the only communication permitted within an AM Request handler is to send at most one AM Reply message to the Request initiator, and AM Reply handlers are not permitted to communicate at all. Message injection is the only blocking operation permitted within a handler. The AM model imposes a number of usage restrictions relative to more general mechanisms such as X10's `async`, which enables it to be efficiently implemented in a deadlock-free manner using a bounded amount of buffering and minimal copying of data. Section 3.2 contains a theorem proving that an Active Message network can be implemented without deadlock and with bounded resource utilization. Though this deadlock-freedom guarantee for Active Messages has been mentioned informally in the past, we are not aware of a prior publication that contains a proof of this result.

The Activity Network in Figure 3 is used to support the more general inter-place interactions required by X10's `async` and `finish` constructs. Each node has an `IN` queue and `OUT` queue of bounded size, R. As discussed in Section 3.3, this resource bound can lead to *physical deadlock* scenarios for certain schedules of X10 programs that are guaranteed to never exhibit a *logical deadlock*.

3.2 Deadlock Freedom in Active Messages

In this section, we show how Active Messages can be implemented with provable deadlock freedom and bounded re-

[2]The term, "active message" has been used in a number of contexts, but for the purposes of this paper we refer to the variant of Active Messages defined by the AM-2 [12] and GASNet [4] specifications.

source utilization by using two *virtual networks*, one for AM Requests and one for AM Replies. Virtual networks [8] provide independent communication channels, ensuring that resource starvation (*e.g.,* lack of buffer space) on one virtual network does not prevent communication progress on the other virtual network. Virtualization is often implemented by multiplexing over a single physical network and maintaining logically independent buffers, but other strategies are possible. The buffer space required by each virtual network is assumed to be fixed and usually established at program initiation.

Generalizing, we define an $AM(N)$ *communication protocol* to be a handler-based communication protocol which delivers messages of a bounded size and is implemented using a fixed number of virtual networks $V_0, \ldots, V_{N-1}$ of increasing priority, and adheres to the following rules:

- All message injection operations executed outside the dynamic context of a handler inject messages on the lowest priority virtual network, V_0.
- A handler running in response to a message received on V_i can only inject messages on the next-higher priority network V_{i+1}.
- Handlers running in response to a message received on the highest-priority virtual network, V_{N-1} may never inject messages.
- Message injection is the only blocking operation permitted within a handler. Handlers are otherwise never permitted to block indefinitely (*e.g.,* to await resources or communication arrival) — they must run to completion independently of other system activities, consuming a bounded amount of resources and issuing a bounded number of message injections.
- Message injection operations block until resources are available to accept the message, and any attempt to inject a message on V_j which cannot complete immediately (*e.g.,* due to resource starvation on that network) must service any incoming messages on $V_j, \ldots, V_{N-1}$ while blocking. Incoming messages are otherwise never serviced from within handler context.
- Each virtual network makes independent progress on message delivery and consumes bounded resources.

Theorem 3.1. *An $AM(N)$ protocol can be implemented without deadlock and with bounded resource utilization.*

See Section A.2 in appendix for proof.

Corollary 3.1.1. *Active Messages can be implemented without deadlock and with bounded resource utilization.*

Proof. Active Messages is just a special case of Theorem 3.1 where $N = 2$:
- All AM Request messages travel on virtual network 0 (low priority network)
- All AM Reply messages travel on virtual network 1 (high priority network)
- AM Request handlers may inject at most one AM Reply, and otherwise may never block.
- AM Request injections poll both networks, AM Reply injections poll only the AM Reply network.
- AM Reply handlers may never inject messages or block. □

3.3 Physical Deadlock and Doppelganger Mode

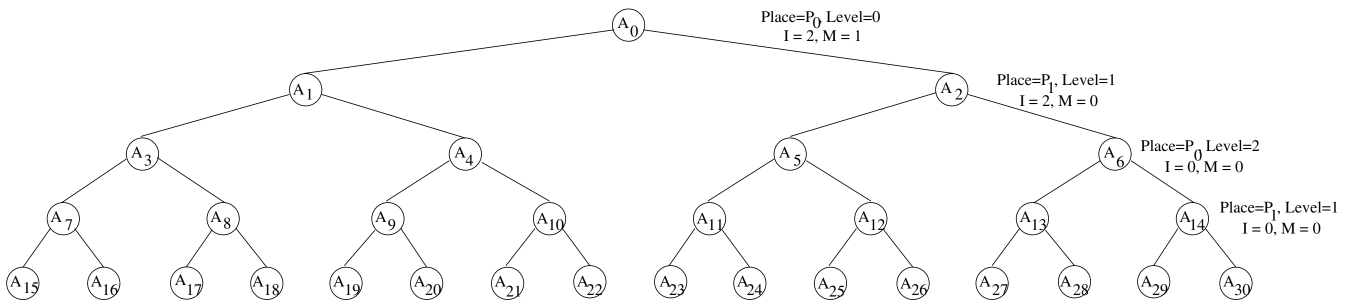
We now move from Active Messages to more general X10 activities. To illustrate the possibility of a physical deadlock with a simple one-to-one mapping from places to nodes, consider the dynamic activity creation tree in Figure 4, for an execution of the X10 code fragment shown in Figure 5 on two places named P_0 and P_1. In this scenario, the `p.next()` operator alternates between places P_0 and P_1. Assume that each node has I slots available in its incoming message queue (each of which can hold one activity while it is being pushed over the network from a peer) and furthermore that each node has M slots in local memory which can hold a stalled activity while it awaits a response from a child activity pushed to a remote node. For the sake of concreteness, assume that $I = M = 2$ in this example.

```
void foo (place p, ...) {
   if (...) {
      ... // recursion termination condition
   } else {
l1:   finish {
l2:      async (p.next()) { foo (here, ...); }
l3:      async (p.next()) { foo (here, ...); }
l4:   }
   }
   ... // Other computation
}
void main () {
l5:   finish async { foo (here, ...); };
}
```

Figure 5: Example X10 Code Fragment for Physical Deadlock Scenario

The dynamic activity creation tree for this X10 program is shown in Figure 4. We assume that this simple deployment has no support for throttling the creation of remote activities. Initially P_0 starts the execution of activity A_0 created in the `main()` method (Statement l5) with 2 empty slots for I and M each. A_0 then creates two child activities at P_1 and subsequently awaits their completion (Statements l1-l4). A_0 is now placed in the Stalled Activity Buffer at place P_0, thereby reducing the number of empty slots for M to 1. At Level 1, P_1 now has 2 incoming activities (A_1 and A_2), each of which again creates 2 child activities at P_0 and waits for their completion. For P_1, the number of empty slots for I is 2, but the number of empty slots for M reduces to 0 (as A_1 and A_2 move to stall buffer). Activities A_3, A_4, A_5 and A_6 created at Level 2 subsequently create 8 activities at P_1. Thus the number of empty slots for I and M now becomes 0 at P_0. These 8 activities recursively create 16 activities at P_1 and make the number of empty slots in I and M also equal 0 at P_1. At this point the system is deadlocked - all incoming queues are full of non-leaf activities, and there is no remaining memory for non-leaf activities to stall while waiting for remote children to terminate. Both nodes are attempting to inject newly-created activities and there is nowhere for them to go, and none of the existing activities can be drained or retired.

Consider an activity, A_i, that was created at place P_j in an X10 program. Note that the physical deadlock scenario in Figure 5 arose for the case of a *standard cluster deployment* in which activity A_i must be scheduled at the physical node, N_j, corresponding to A_i's place. We will now consider a *modified cluster deployment* in which activity A_i is executed on some other physical node, $N_k \neq N_j$, while still preserving the illusion that A_i is executing at its designated



Place=P_0 Level=0
I = 2, M = 1
Place=P_1, Level=1
I = 2, M = 0
Place=P_0 Level=2
I = 0, M = 0
Place=P_1, Level=1
I = 0, M = 0
A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11} A_{12} A_{13} A_{14} A_{15} A_{16} A_{17} A_{18} A_{19} A_{20} A_{21} A_{22} A_{23} A_{24} A_{25} A_{26} A_{27} A_{28} A_{29} A_{30}

Figure 4: Dynamic Activity Creation Tree for Physical Deadlock Scenario

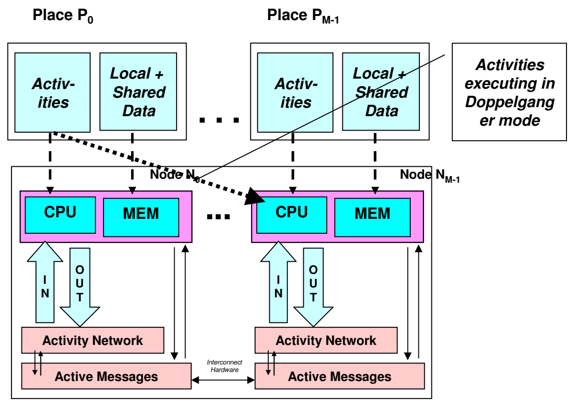


Figure 6: Cluster Deployment with Doppelgänger Mode; One-to-many mapping of activities and one-to-one mapping of data from places to nodes

place. As shown in Figure 6, this kind of deployment works by allowing activities to execute in *Doppelgänger Mode*. As indicated by the dotted arrow in Figure 6, when an activity executes in Doppelgänger Mode, it is permitted to run on a different node from the one where its place's data is located. Doppelgänger Mode only impacts the mapping of activities to nodes; the mapping of data from places to nodes remains unchanged from Figure 3 to Figure 6. Note that the mapping of activity A_i to remote node N_k is a deployment step that does not impact the logical identity of A_i's place in the X10 program. In particular, the expression, `here`, will continue to refer to place P_j for activity A_i, even when A_i is executing on node N_k. Further details on Doppelgänger mode are discussed in Section 3.3.

We now briefly discuss the four cases that can lead to inter-node communication in Doppelgänger mode:

Place-local data accesses (Case 1).

All place-local data accesses in A_i executing at node N_k (*i.e.,* accesses to data in place P_j, which will be located at node N_j) will have to be performed remotely via the Active Message network since A_i is not running on node N_j. For example, a read operation on an object field `p.x` or array element `A[i]` that could have been performed locally if A_i was executing on N_j will now have to be performed remotely. Since the extra number of remote messages may degrade performance, the heuristic algorithm in Section 4 uses Doppelgänger mode as a last resort for deadlock avoidance. Note that all activity-local data for A_i can be allocated and accessed at node N_k without requiring any remote communication.

Activity creation (Case 2).

When A_i creates a child activity, the destination place (P_x, say) for the activity is specified by the X10 program's semantics. The heuristic algorithm in Section 4 will attempt to create the activity at node N_x. However, if resource constraints do not permit this inter-node communication for activity creation, then the heuristic algorithm will instead create the activity at node N_k so that it too will execute in Doppelgänger mode if $N_x \neq N_k$.

Finish notification (Case 3).

If activity A_i enters a `finish` statement, the fact that A_i is executing on node N_k will be propagated to all descendant activities created in the scope of the `finish` so that their termination notifications will be sent to node N_k rather than node N_j.

Atomic blocks (Case 4).

Atomic blocks need special handling in Doppelgänger Mode. If activity A_i contains an `atomic` statement, the implementation needs to ensure that its combination of operations on activity-local data (on node N_k) and place-local data (on node N_j) is performed atomically. This can be accomplished by the following steps:

1. **Copy-in:** Activity A_i makes a copy of all activity-local variables that may be read in the atomic statement, for transmission from node N_k to node N_j.
2. **Active Message request:** A_i then sends a request message to node N_k with a copy of the local variables from Step 1 and a pointer to the code for the `atomic` statement, and blocks waiting for the reply. Atomic statements in X10 are guaranteed to satisfy all the constraints for active message handlers outlined in Section 3.2, including no place-remote accesses.
3. **Active Message reply:** After completion of the active message handler for the `atomic` statement, node N_j sends a reply to node N_k with a copy of all activity-local variables that may have been written in the `atomic` statement.
4. **Copy-out:** Activity A_i copies values of the local variables received in the reply message, and then resumes execution.

4. DEADLOCK-FREE SCHEDULING OF MULTI-PLACE X10 COMPUTATIONS

The multi-place scheduling algorithm (outlined in Figure 7) is a novel distributed algorithm that governs the scheduling of X10 activities at multiple places, while maintaining provable bounds on resource utilization at each place. In this paper, we restrict our attention to the case where there is exactly one processor per place, which corresponds to a cluster of uniprocessors. Furthermore, for simplicity, we assume the multi-place algorithm never performs work-stealing across the cluster — once created, activities run to completion on a given processor, although that processor may or may not correspond to the place expression (depending on the decisions made by the scheduling algorithm at activity creation). Future work includes expanding the multi-place algorithm to allow work-stealing while still maintaining provable resource utilization bounds per place. The complete details of the algorithm cannot be covered here due to space restrictions, so we shall focus on the portions relevant to the proof of bounded resource utilization.

X10 supports dynamic creation of activities, and in general the values of the place expressions associated with activity creation are only discovered at runtime as the computation unfolds – therefore scheduling decisions are made online using a greedy heuristic. We shall use the terms *local activity* and *remote activity* to indicate an activity which is spawned with a place expression which respectively equals or differs from the place of the processor executing the parent activity.

By default, the heuristic assigns activities to the processor hosting their semantically-specified place – local activities are always executed on the local place, and remote activities are pushed to their target place upon creation. Once a remote activity is successfully pushed to a remote place, the local processor is free to continue executing the parent activity and servicing its local deque, as the depth-first exploration of the remote activity subtree has effectively been handed off to the remote processor. When there is resource contention at the source or target place, the heuristic exercises flow control by running remote activities on the local processor in Doppelgänger mode.

As in the single-place scheduling algorithm, each processor has an associated ready deque of activities which govern the order in which the activities are processed. One important difference we shall see is this deque allows for insertion at a well-defined point in the middle, although this paper will still refer to the data structure as a deque for consistency with existing literature. Except where otherwise noted, the processor services the deque as described in section 2.3 – in the absence of remote `async` operations, activities are serviced by the processor in depth-first order.

Additionally associated with each processor is a *stalled activity buffer (SAB)*, an area of memory which holds activities that have stalled upon a `finish` operation waiting for the completion of a descendant activity. Due to the depth-first nature in which local activities are processed, an activity can only enter the SAB if it has stalled awaiting the completion of a descendant activity that has been pushed to a remote processor, but not yet completed.

At all times, every live activity in the system will reside in exactly one of three areas: the processor (which executes at most one activity at a time), the deque (where it awaits servicing by the processor) or the SAB (where a stalled activity awaits to be reenabled by one or more remote descendant activities). The scheduling algorithm manages the movement of activities between these locations at each node, and the system resource utilization at each node can be captured by summing the contributions of activities in each area.

4.1 Useful Notations

Some useful notations as used by the algorithm in Figure 7 are defined here.

- A_i where $i \in \{1, 2, \ldots\}$ stands for an activity.
- *finish*(A_i) denotes the activity that contains the `finish` instruction to which A_i must report completion.
- P_i denotes a place, N_i denotes a node.
- *node*(A_i) denotes the node hosting the activity A_i. *node*(P_i) denotes the node hosting the place P_i. *deque*(N_i), $SAB(N_i)$ denote the deque and Stalled Activity Buffer respectively for node N_i.
- $Count\_in(N_i)$ denotes the counter which keeps track of the bound R on fresh activities in the deque. $Count\_out(N_i)$ checks for the bound R on the stalled activities in the SAB.
- The predicate $Fresh(A_i)$ is set if A_i is a fresh activity and it is initialized to $false$;

4.2 Algorithm Outline

The algorithm in Figure 7 is based on the following ideas. Consider an activity A_1 at place P_j executing at node N_j. At an execution step, any one of the following is possible.

1. A_1 creates a local activity A_2 at P_j. Following the depth-first execution rule, A_1 will be pushed to bottom of the local deque at N_j and A_2 will start executing.
2. A_1 creates a remote activity A_2 to be executed at P_r (where $r \mathrel{!}= j$). There are two possibilities in this case: either the system has enough resources for N_r to accept A_2 or it does not have. In case it has enough resources, A_2 is pushed to the top of N_r's deque. Otherwise, A_2 is executed at N_j in Doppelgänger mode.
3. A_1 stalls because of a `finish` instruction which is waiting for a descendant activity to complete. Note that due to the depth-first processing of local activities, this can only occur when some descendant was pushed to a remote node. In this case, A_1 is moved to the SAB.
4. A_1 completes its execution. It sends a completion notification to activity $A_2 = finish(A_1)$. If A_1 was the last dependency for the `finish` in A_2, then A_2 is enabled and moved out from SAB to the *middle-insertion point* of the deque at the corresponding processor.
5. A_1 performs a finite computation.

It is clear that at any node, the deque and SAB structures are manipulated for managing the activities. Therefore, to show that the X10 computation needs bounded resources assuming bounded S_1 and S_{max} as defined in section 2, we should show that we will never require the deque or the SAB to be unbounded. In the following, we outline the ideas incorporated in the algorithm to keep the deque and the SAB bounded.

- A deque at any node consists of activities that have been either created locally or pushed by remote nodes.

In case of locally created activities, they follow depth-first execution rules thus ensuring that a bounded deque suffices for local activities. However, we need to limit the activities being pushed by remote nodes, otherwise remote nodes could go on pushing activities without regard to resource limitations at the target. The way to handle incoming activities from remote nodes is that the activities in the deque are distinguished as *fresh* or *worked-on* based on the follow definition: an activity is considered fresh if it has not moved to the processor even once, otherwise it is considered worked-on. The *middle-insertion point* is defined to be the point below the bottommost *fresh* activity (if any) and the topmost *worked-on* activity (if any). The activities from the SAB that are inserted at this middle-insertion point fall in the class of worked-on activities. Incoming activities are accepted if and only if the number of fresh activities is less than or equal to R, where R is a predefined constant positive integer. In case an activity is not accepted by this node, then the activity will run in Doppelgänger mode at the remote source node. This is taken care of by parts (B) and (D) of the algorithm in Figure 7.

- The SAB for a node contains the stalled activities, and therefore if we do not limit the movement of activities to SAB, we may need an unbounded SAB. To bound the SAB, we bound with R the number of activities that are waiting for the completion of remote activities (activities that were pushed to remote nodes). Note that they should not be waiting for completion of any local activity. These activities exactly correspond to those activities rooting a spawn subtree that executed one or more remote push operations and contain the innermost `finish` instruction for those remote pushes. Whenever there are R such activities in SAB and a remote activity is created within a local `finish` scope, then the node resorts to Doppelgänger mode. This is incorporated in parts (A), (C), and (F) of the algorithm in Figure 7.
- Finally, we have to consider the case when an activity is moved to the middle-insertion point of the deque from SAB after getting enabled. The decision of putting the activity at the middle-insertion point ensures that the deque cannot process any fresh activity until all the worked-on activities at that processor have either stalled or completed. This ensures no extra demand on the resources deque or SAB.

The only thing left to be shown is that no deadlock can arise, if we hit the resource bounds by following the multi-place algorithm as in Figure 7. As we have seen, whenever the bound R is reached on deque or SAB, it is appropriately handled by using the Doppelgänger mode where deadlock-free Active Messages are used to perform any place-local data accesses using fine-grained messages. Thus progress is always ensured and no deadlock can arise in spite of the bounded resource setup.

THEOREM 4.1. *The multi-place algorithm in Figure 7 uses at most* $2 * R * S_{max} + R * S_1 + S_1$ *bytes of space per node and ensures deadlock freedom.*
See Section A.3 in appendix for proof.

5. RELATED WORK

While UPC [16], Titanium [10] and Co-Array Fortran [14] follow an SPMD style control-flow model where all threads execute the same code [9], Cilk [7], Chapel [11], Fortress [1] and X10 choose a more flexible model and allow more general forms of control flow among concurrent activities. This flexibility removes the implicit co-location of data and processing that is characteristic of the SPMD model. X10's `async` and `finish` are conceptually similar to Cilk's `spawn` and `sync` constructs. However, X10 is more general than Cilk in that it permits a parent activity to terminate while its child/descendant activities are still executing, thereby enabling an outer-level `finish` to serve as the root for exception handling and global termination. Fortress offers library support that enables the runtime system to distribute arrays and to align a structured computation along the distribution of the data. X10 is somewhat different, in that it introduces *places* as a language concept and abstraction used by the programmer to explicitly control the allocation of data and processing. A place is similar to a locale in Chapel and a site in Obliq [5]. Like X10, each object in Obliq is allocated and bound to a specific site and does not move; objects are accessed though "network references that can be transmitted from site to site without restrictions" [5]. In Obliq, arrays cannot be distributed across multiple sites. Chapel's model of allocation is different from X10 and Obliq because Chapel does not require that an object be bound to a unique place during its lifetime or that the distribution of an array remain fixed during its lifetime.

The Cilk-NOW project [2] explored the use of work-stealing algorithms in a distributed-memory environment, with adaptive parallelism and fault-tolerance. In that system task migration was entirely pull-based (via a randomized work stealing algorithm), in contrast to our algorithm which pushes work to specific places guided by linguistically-explicit locality expressions (which Cilk lacks). To the best of our knowledge, no formal proof for the deadlock-freedom or resource utilization properties in the distributed version of Cilk has been published thus far.

The concept of Active Messages was first introduced in [17] and later standardized in the AM-2 specification [12], and the ideas have subsequently appeared in various forms in a number of systems. One recent example is the GASNet communication system [4], which serves as the communication layer for several global-address-space language compilers. GASNet includes an Active Message interface modeled on AM-2, and augmented with extensions to allow handler concurrency through explicit atomicity control (handler-safe locks and no-interrupt sections). The deadlock-freedom guarantee from Section 3.2 can be extended to other, more general variants of Active Messages as well.

There is extensive literature on deadlock-freedom in routing algorithms for interconnect hardware and the use of virtual networks - see [13] for a good survey. Proofs of deadlock-freedom in routing algorithms rely upon the assumption that messages arriving at a destination node are eventually consumed – our proof can be seen as a verification that this property is always maintained under an $AM(N)$ protocol, which still maintains bounded resource utilization.

```
(A) Activity $A_1$ at node $N_1$ creates activity $A_2$:
  1. $FA := finish(A_2)$; $FN := node(FA)$;
  2. if $node(A_2) == N_1$ then // local activity
     (a) Create $A_2$ at $N_1$. Push $A_1$ to bottom of $deque(N_1)$ and start executing $A_2$.
     else // remote activity
     (a) // Initialize $doPush$ to true if local conditions permit a remote $push$ operation for creating $A_2$
         $doPush := (FN \neq N_1) \mid\mid hasRemoteChildren(FA) \mid\mid (Count\_out(N_1) < R)$;
     (b) if $doPush$ then
         i. Send $AM\ request$ to create $A_2$ as a fresh activity on $node(A_2)$
         ii. Block to await $AM\ reply$; $(see\ (B))$
         iii. if reply == failure then $doPush :=\ false$;
     (c) if $doPush$ then $hasRemoteChildren(FA) := true$; // Remember a child activity was pushed to remote node else Execute
         activity $A_2$ at node $N_1$ in Doppelgänger mode
(B) Node $N_2$ receives an AM request to create $A_2$:
  1. if $Count\_in(N_2) < R$ then
     (a) $Count\_in(N_2)$++ ; $Fresh(A_2) := true$ ; Put activity $A_2$ at the top of $deque(N_2)$ ;
     (b) Send $success$ reply in response to $AM\ request$.
     else Send $failure$ reply in response to $AM\ request$.
(C) Activity $A_1$ at $N_1$ stalls to wait for descendants to complete:
  1. if (hasRemoteChildren($A_1$)) then $Count\_out(N_1)$++; // $A_1$ is innermost finish for remote push
  2. Move $A_1$ to SAB($N_1$).
(D) Activity $A_2$ is removed from the deque by the processor to work on:
  1. if $Fresh(A_2)$ then $Count\_in(node(A_2))$--; // Allow more incoming activities to be accepted.
  2. $Fresh(A_2) := false$;
(E) Activity $A_2$ completes, and is the last living local activity in a subtree pushed from a remote node:
  1. $FA := finish(A_2)$; $FN := node(FA)$;
  2. Send completion notification for $A_2$ to activity $FA$ on node $FN$.
(F) Activity $FA$ at Node $FN$ receives completion notification for $A_2$:
  1. if $FA$ is enabled then
     (a) hasRemoteChildren($FA$) := $false$; //$FA$ is enabled means that all its descendants have completed.
     (b) Move $FA$ to middle-insertion point of deque($FN$) ;
         When the processor next picks up $FA$ to work on, it will do $Count\_out(FN)$--
```

Figure 7: Multi-place Deadlock-free Heuristic Scheduling Algorithm

6. CONCLUSIONS

X10 provides a general dynamic parallelism model in the setting of a virtualized partitioned memory model, and therefore gives programmers the ability to have dynamic parallelism along with locality control. The X10 runtime system is responsible for resource management and, as shown in this paper, there are potential pitfalls that can lead to physical deadlock due to unbounded resource consumption.

We addressed the problem of guaranteeing the absence of *physical deadlock* in the execution of a parallel program using the X10 `async`, `finish` and `atomic` constructs. We extended previous work-stealing memory bound results for the Cilk language [3] for a single-place execution on an SMP to admit more general synchronization patterns in which one activity may stall for completion of a descendant activity, not just an immediate child as in Cilk. We introduced a new heuristic multi-place algorithm (with a Doppelgänger mode) for deployment on a cluster of uniprocessors that guarantees the absence of physical deadlock in the presence of bounded communication resources. While several language groups have proposed constructs for combining locality control with dynamic threading, we believe that this paper contains some of the first results guaranteeing deadlock-free execution in the presence of bounded resources. For future work, we plan to combine both results to obtain a physical-deadlock-freedom guarantee for deployments on clusters of SMPs.

7. REFERENCES

[1] Eric Allan, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 0.618. Technical report, Sun Microsystems, April 2005.

[2] Robert D. Blumofe. *Executing multithreaded programs efficiently*. PhD thesis, Cambridge, MA, USA, 1995.

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

[4] Dan Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[5] Luca Cardelli. A language with distributed scope. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'95)*, pages 286–297, January 1995.

[6] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track*, 2005.

[7] Cilk-5.3 reference manual. Technical report, Supercomputing Technologies Group, June 2000.

[8] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.*, 36(5):547–553, 1987.

[9] Frederica Darema, David A. George, V. Alan Norton, and Gregory F. Pfister. A Single-Program-Multiple-Data Computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[10] Paul Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.

[11] Cray Inc. The Chapel language specification version 0.4. Technical report, Cray Inc., February 2005.

[12] Alan Mainwaring and David Culler. Active message applications programming interface and communication subsystem organization. Tech Report UCB/CSD-96-918, U.C. Berkeley, 1995.

[13] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26:62–76, 1993.

[14] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. In *ACM Fortran Forum 17, 2, 1-31.*, 1998.

[15] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'05)*, August 2005.

[16] UPC language specifications, v1.2. Technical Report LBNL-59208, Berkeley National Lab, 2005.

[17] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

A. APPENDIX: TECHNICAL DETAILS

A.1 Deadlock Free Scheduling for Single Place X10 Computations (Section 2)

LEMMA A.1. *In the execution of any terminally-strict (X10) computation by the Work-Stealing Algorithm, let us consider any processor p and at the beginning of a given time step let there be k number of activities in p's ready deque. Let Γ_0 be the activity that p is working on. Let $\Gamma_1,\ \Gamma_2, \ldots \Gamma_k$ denote the k activities ordered from bottom to top, that is, Γ_1 is the bottommost and Γ_k is the topmost. If $k > 0$, then the activities in p's ready deque satisfy the following properties:*

1. *For $i = 1, 2, \ldots, k$, activity Γ_{i-1} is the child of Γ_i.*
2. *For $i = 1, 2, \ldots, k$, activity Γ_i has not been worked upon since it spawned Γ_{i-1}.*

PROOF. We focus on the case where a stalled activity is enabled in a terminally-strict (X10) computation, since the other cases have already been addressed in the proof of Lemma 4 in [3].

If Γ_0 enables a stalled activity Γ_x, then either $k = 0$ or the bottommost activity in p's ready deque is Γ_x. This is because an activity can be enabled only when all the dependencies on descendants are resolved. A dependency is resolved only when the activity from which there is a dependence edge completes (dies). Thus, if Γ_0 enables Γ_x, it means that all ancestors, if any, of Γ_0 between itself and Γ_x have completed. If Γ_0 and Γ_x are in same p and Γ_0 enables Γ_x, then by induction we conclude that Γ_x must be the parent of Γ_0 and also the bottommost. If Γ_0 and Γ_x do not belong to same p and Γ_0 enables Γ_x, then by induction the deque should have $k = 0$. These two statements imply that either $k = 0$ or Γ_x is parent and bottommost. Note that since Γ_0 can enable Γ_x only after completing its last instruction (due to terminally-strict property), by the end of this time step, Γ_x would have been brought to p according to the rules of the Work-Stealing Algorithm.

These conditions on the deque derived for terminally-strict computation exactly correspond to that for fully-strict computation. Therefore, we claim that the Property (1) and (2) of the lemma are preserved for terminally-strict computations in the case when an activity is enabled. □

LEMMA A.2. *The Work-Stealing Algorithm for terminally-strict X10 computations preserves the busy-leaves property.*

PROOF. The proof is straightforward induction on execution time. Let $k, \Gamma_0, \Gamma_1, \ldots, p$ be as defined in Lemma A.1. For k=0, the lemma is vacuously true at the outset. Let Γ'_0 denote the activity being worked on by p after the step. Let $\Gamma'_1, \Gamma'_2 \ldots \Gamma'_{k'}$ denote the k' activities in p's ready deque after the step. We now use Lemma A.1 and the rules of Work-Stealing Algorithm to prove that lemma is preserved. That is, we will show either $k' = 0$ or the property holds.

If Γ_0 spawns a child, then p pushes Γ_0 onto the bottom of the ready deque and commences work on the child. Γ'_0 denotes the child which is being worked upon. By Lemma A.1, Γ'_0 is the leaf node of the current spawn subtree and thus busy-leaves property is maintained.

If Γ_0 stalls, then we know that Γ_0 was the only activity running on processor and $k = 0$. Therefore, the processor commences work stealing. The busy-leaves property is maintained by default.

If Γ_0 dies, then we have to consider two cases. If $k = 0$, then the ready deque is empty, so the processor commences work stealing and when it steals and commences work on an activity, we have $k' = 0$. Thus the lemma holds in this case. If $k > 0$, the ready deque is not empty, so the processor pops the bottommost activity off the deque and commences work on it. Thus, by virtue of being bottommost, we can deduce from Lemma A.1, that this is the leaf node of current spawn subtree and thus the busy-leaves property holds.

If Γ_0 enables a stalled activity Γ_x, then either $k = 0$ or the bottommost activity in p's ready deque is Γ_x. This is because an activity can be enabled only when all the dependencies on descendants are resolved. A dependency is resolved only when the activity from which there is a dependence edge completes (dies). Thus, if Γ_0 enables Γ_x, it means that all ancestors, if any, of Γ_0 between itself and Γ_x have completed. This in conjunction with Lemma A.1 implies that either $k = 0$ or Γ_x is bottommost.
If Γ_x is the bottommost activity in p's ready deque, this implicitly means that Γ_0 was the last descendant with a dependence edge to Γ_x's next instruction. Thus, when Γ_x is enabled it does not have any living descendants and thus the busy-leaves property holds true. If $k = 0$, then the enabled activity Γ_x is brought to the p's ready deque and made the bottommost in the deque. This means that Γ_x starts executing from next step, thus maintaining the busy-leaves property.

It is clear that all living activities for which there are no living descendants are getting worked upon by some processor. Thus we prove the busy-leaves property. □

A.2 Deadlock Freedom in Active Messages (Section 3.2)

THEOREM A.3. *An $AM(N)$ protocol can be implemented without deadlock and with bounded resource utilization.*

PROOF. By definition, deadlock occurs when two or more processes are blocked with no ability to make progress. The only blocking operation permitted in an AM(N) protocol is message injection, therefore it suffices to prove deadlock cannot occur when two or more processes are blocked while injecting messages.

The proof is by induction on decreasing virtual network number, with the base case of V_{N-1}. All message injection operations that block must service incoming messages on V_{N-1}, and handlers running in response to messages received on V_{N-1} must execute and complete without consuming additional messaging resources. Therefore, by the independent progress guarantee of virtual networks, V_{N-1} is guaranteed to eventually make progress, allowing injection operations on V_{N-1} to eventually succeed, proving the base case. For the inductive case, handlers executing in response to messages received on V_i may only block while injecting a message on V_{i+1}. By the inductive hypothesis, that injection operation will eventually succeed allowing the handler to make progress towards completion, ensuring that it will eventually retire and allow progress on injections into V_i. Therefore, all message injection operations are guaranteed to eventually make progress, implying deadlock cannot occur.

The resource utilization of an $AM(N)$ protocol is comprised of the resources needed to implement virtual networks $V_0, \ldots, V_{N-1}$, plus the resources needed for any handlers simultaneously active on the program stack of each process. The resource utilization of each virtual network is assumed to be bounded at a fixed value. Handlers for messages on V_i inject messages on V_{i+1} and therefore may only be preempted to service incoming messages on higher-priority networks $V_{i+1}, \ldots, V_{N-1}$ — therefore at most N handlers may be active on the program stack at any time. Each such handler consumes bounded resources, therefore the total resource utilization of the $AM(N)$ protocol is bounded. □

A.3 Space Bounds for Multi-Place Algorithm (Section 4)

LEMMA A.4. *A fresh activity is picked by a processor at a node from the deque only when all the worked-on activities in that node until that instant are either completed or stalled.*

PROOF. To begin with, let us recall what is a *fresh* activity. A fresh activity is a term associated with an activity that has been pushed on the deque by a remote node and has not been executed by the processor even once. When an activity is picked by a processor to work on, then the spawn subtree rooted at this activity is explored in a depth first manner. Note that the activities created locally during the course of processing the spawn subtree are regarded as worked-on and not fresh. The activities in SAB automatically fall in the class of worked-on by definition.

Suppose at some instant in time at a node n, the deque has some fresh activities denoted by set $\mathcal{F}$ and some worked-on activities denoted by $\mathcal{W}$. Note that fresh activities are pushed onto the top of the deque. Therefore, the set $\mathcal{F}$ is placed above the set $\mathcal{W}$ in the deque. Another point to note is that the activities in SAB upon getting enabled are inserted between $\mathcal{F}$ and $\mathcal{W}$ in the deque and are added to the set $\mathcal{W}$.

Let us now see the progress at node n from this instant onwards. As mentioned earlier, the processor picks activities from the bottom of deque which implies that in our case it will pick an activity w, where $w \in \mathcal{W}$. The case where $\mathcal{W}$ is empty is similar to Case 1 below. The case $\mathcal{F} = \phi$ vacuously holds for all the following cases. Let $\mathcal{W}' = \mathcal{W} \setminus w$. The possible cases that may arise are as follows:

1. w completes and $\mathcal{W}' = \phi$ and no activity moves to deque from SAB: In this case, an activity in $\mathcal{F}$ becomes bottommost in the deque and is picked by the processor.
2. w completes and $\mathcal{W}' \neq \phi$: In this situation, the bottommost activity will be from $\mathcal{W}'$. Hence, a worked-on and not fresh activity will be picked by the processor.
3. w stalls and $\mathcal{W}' = \phi$ and no activity moves to deque from SAB: Here, w is moved to SAB and an activity from $\mathcal{F}$ is picked to work on.
4. w stalls and $\mathcal{W}' \neq \phi$: In this case, w is moved to SAB and an activity from $\mathcal{W}'$ is picked to work on.
5. w spawns a child say w_c: The activity w is pushed to the bottom of deque and added to set $\mathcal{W}$ if it is not a member already. The processor starts executing the child activity w_c.

All the cases above show that a fresh activity is picked by a processor only when the worked-on activities in deque get completed or stalled. □

LEMMA A.5. *The algorithm requires at most $R * S_1$ bytes of space per node for the SAB.*

PROOF. For a node N, the value of Count_out(N) is incremented whenever an activity has to stall waiting for completion of its descendants at remote nodes. Note that Count_out never exceeds R as can be seen from the algorithm. This implies that at any point in time, we can have at most R activities that are waiting for completion of their remote descendants.

A stalled activity in SAB may also have its ancestors in that SAB. Suppose A is an activity in the SAB at N such that it has some ancestors in the SAB too. Then the oldest such ancestor in the SAB will be referred to as root of A or root(A). The ancestors may also have immediate remote children in which case they will contribute to Count_out, otherwise if they are in SAB because of their descendants that are waiting for remote children to finish, then they do not contribute to Count_out.

As we have seen, an upper bound of R has been enforced on the number of activities in N that have remote descendants. The fact that Count_out gets decremented only when an enabled activity that had moved to deque reaches the processor ensures this bound. We now have to bound the activities that reside in the SAB but do not have remote descendants. We have the following cases to consider:

1. *Count_out $= R$ and for all R activities in SAB at N, their root is different:* In the worst possible case, the stack depth for each of the R activities in SAB is S_1. (since the stack depth can have the maximum size of S_1 bytes by definition.) Since Count_out is R, no new activities will be moved to SAB as per the algorithm

and, therefore, the upper bound in this case is $R * S_1$ bytes.

2. *Count_out* $= R$ *and some activities among the* R *activities share their root.* In this case, the total number of root activities for these R activities will be less than R. Then by an argument similar to Case 1, it is clearly seen that the upper bound on the size of SAB is smaller than $R * S_1$ bytes.
3. *Count_out* $< R$: It is straightforward to see in this case that the size of SAB is less than $R * S_1$ bytes.

These cases cover all possibilities and therefore, the upper bound on size of SAB at a node is $R * S_1$ bytes. □

LEMMA A.6. *The size of the worked-on activities in the deque is bounded by* $S_1 + R * S_{max}$ *bytes of space per node.*

PROOF. According to the algorithm, there are three sources of worked-on activities in the deque at a node:

1. SAB
2. The activities spawned by worked-on activities are also considered worked-on.
3. A fresh activity gets picked by a processor and, therefore, its status changes to worked-on.

Let us first consider the source of SAB. Any activity that moves to deque from SAB after getting enabled, is pushed at the top of current set of worked-on activities at that node. These enabled activities can enable their ancestors in SAB only when they are picked by processor at that node. We know that a SAB can have a maximum of $R*S_1$ size, that is at most R leaf activities. This implies that there can be no more than R such activities in the deque that have moved from SAB to deque but have not been picked by processor. This implies that the size required by this set of activities is bounded by $R * S_{max}$.

Now let us consider the remaining two sources (2 and 3) above. The processor picks the bottommost activity in the deque. It can be either a worked-on activity or a fresh activity in case the condition of lemma A.4 is met.

During exploration of the spawn subtree rooted at bottommost activity, the size of deque can grow by at most S_1 bytes which is the maximum possible depth of the spawn subtree. The Count_out value at the instant when exploration begins decides the number of activities that can be pushed to SAB. If Count_out value reaches R, then the node switches to Doppelgänger mode. Thus, we see that in all situations, sources 2 and 3 above require a maximum of S_1 bytes.

We thus conclude that the space required by worked-on activities is bounded by $S_1 + R * S_{max}$. □

THEOREM A.7. *The multi-place algorithm in Figure 7 uses at most* $2 * R * S_{max} + R * S_1 + S_1$ *bytes of space per node and ensures deadlock freedom.*

PROOF. We need to account for following activities in a node:

1. Fresh activities in deque
2. Worked-on activities in deque and SAB

There is an upper bound of R on the number of fresh activities in a deque and the size of each activity has been bounded by S_{max}. Therefore, the space required by fresh activities in the deque is bounded by $R * S_{max}$ bytes.

Let us consider the worked-on activities now. As shown in lemma A.6, they are bounded by $S_1 + R * S_{max}$. According to Lemma A.5, they are bounded by $R * S_1$ in SAB. Hence, we prove the total bound of $2 * R * S_{max} + R * S_1 + S_1$ per node.

The multi-place scheduling algorithm ensures freedom from deadlocks that may arise due to resource bounds in case of normal scheduling. This assurance comes from the Doppelgänger mode by which we ensure that a node does not wait on another node to accept its remote activity. Since there is no such wait involved at any node, an activity that has been spawned already is guaranteed to complete either normally or by using Active Messages (which have also been shown to be deadlock free). □