RI09010 July 22, 2009 Computer Science

IBM Research Report

Affinity Driven Distributed Scheduling Algorithms For Parallel Computations

Shivali Agarwal, Ankur Narang

IBM Research Division
IBM India Research Lab
4, Block C, Institutional Area, Vasant Kunj
New Delhi - 110070. India.

Rudrapatna K. Shyamasundar

Tata Institute of Fundamental Research
Homi Bhabha Road
Mumbai - 400005. India.

Affinity Driven Distributed Scheduling Algorithms for Parallel Computations

Shivali Agarwal, Ankur Narang
{shivaaga, annarang}@in.ibm.com
IBM India Research Laboratory, New Delhi, India

Rudrapatna K. Shyamasundar
shyam@tifr.res.in
Tata Institute Of Fundamental Research, Mumbai, India

Abstract

With the advent of many-core architectures efficient scheduling of parallel computations for higher productivity and performance has become very important. Distributed scheduling of parallel computations on multiple *places* [1] needs to ensure *physical* (due to resource dependency cycle) deadlock free execution along with efficient space-time trade-offs. This makes distributed scheduling particularly challenging. This report presents two algorithms for affinity driven distributed scheduling of *multi-place* parallel computations with physical deadlock freedom.

We first present an online affinity driven distributed scheduling algorithm for *strict* place annotated multi-threaded computations that assumes unconstrained space. We show that the lower bound on the expected execution time is $O(\max_k T_1^k/m + T_{\infty,n})$ and the upper bound is $O(\sum_k (T_1^k/m + T_\infty^k))$; where k is a variable that denotes places from 1 to n, m denotes the number of processors per place, T_1^k denotes the execution time for place k using a single processor and $T_{\infty,n}$ denotes the execution time of the computation on n places with infinite processors per place. Further, we derive bounds on expected message complexity and probabilistic lower and upper bounds on time and message complexity for this scheduling algorithm.

Next, we present a novel affinity driven online distributed scheduling algorithm assuming bounded space per place. If the input application has no logical deadlocks due to control, data or synchronization dependencies then this scheduling algorithm guarantees deadlock free execution using distributed deadlock avoidance strategy. This distributed scheduling algorithm is designed for *terminally strict* parallel computations. We prove that the lower bound on the time complexity of this algorithm does not deviate by more than $\log(D_{max})$ factor (where D_{max} is the maximum depth of the activity computation tree) compared to the unconstrained

[1]place is a group of processors with shared memory

space case. We also present proof for distributed deadlock freedom. To the best of our knowledge, this is the first time affinity driven deadlock-free distributed scheduling algorithms have been presented and analyzed for space and time and message bounds under both unconstrained space and bounded space.
Keywords : Distributed Scheduling algorithm, performance, work-stealing, distributed deadlock avoidance, multi-place parallel computation, strict computation, terminally strict computation, active message network, asymptotic time complexity.

Work Stealing; Scheduling; Multithreaded Computation; Algorithm

1 Introduction

With the advent of multi-core and many-core architectures scheduling of parallel programs for higher productivity and performance has become an important problem. Languages such X10 [7], Chapel [2] and Fortress [3] which are based on partitioned global address (PGAS [4]) paradigm, have been designed and implemented as part of DARPA HPCS program [5] for higher productivity and performance on many-core and massively parallel platforms. These languages have in-built support for initial placement of parallel programs and therefore data locality comes implicitly with the programs. The run-time system of these languages needs to provide algorithmic online scheduling of parallel computations with medium to fine grained parallelism. For handling large parallel computations the scheduling algorithm should be designed to work in a distributed fashion on many-core and massively parallel architectures. Further it should ensure *physical* deadlock free execution under bounded space. It is assumed that the parallel computation does not have any logical deadlocks due to control, data or synchronization dependencies, so deadlocks(referred to as *physical* deadlocks) can only arise due to cyclic dependency on bounded space. This is a very challenging problem since the distributed scheduling algorithm needs to provide *efficient space and time complexity* along with *distributed deadlock freedom.*

The two affinity driven distributed scheduling problems we address are as follows: Given, **(a)** An input computation DAG that represents a parallel computation with fine to medium grained parallelism and mapping from each node to a *place*; **(b)** A cluster of n SMPs(each SMP [6] also referred to as *place*, has fixed number(m) of processors and memory) as the target architecture on which to schedule the computation DAG. For both problems one needs to generate a schedule for the nodes of the computation DAG in an online and distributed

[2]chapel.cs.washington.edu

[3]http://research.sun.com/projects/plrg/

[4]http://x10-lang.org/

[5]www.highproductivity.org

[6]Symmetric MultiProcessor: group of processors with shared memory

fashion that ensures exact mapping of nodes onto *places* as specified in the input DAG. Specifically, for the first problem we assume that the input is a *strict* (section 2) computation DAG and there is unconstrained space per place. Here, we need to design a distributed scheduling algorithm that computes an online schedule for the nodes in the computation DAG while minimizing the time and message complexity. For the second problem we assume that the input is a *terminally strict*(section 2) parallel computation DAG and the space per place is bounded. Here, the aim is to ensure physical deadlock free execution while keeping low time and message complexity for execution.

Scheduling of dynamically created tasks for shared memory multi-processors has been a well studied problem. The work on Cilk [5] promoted the strategy of *randomized work stealing*. Here, a processor that has no work (*thief*) randomly steals work from another processor (*victim*) in the system. [5] proved efficient bounds on space ($O(P \cdot S_1)$) and time ($O(T_1/P + T_\infty)$) for scheduling of *fully-strict* (section 2) computations in an SMP platform; where P is the number of processors, T_1 and S_1 are the time and space for sequential execution respectively, and T_∞ is execution time on infinite processors. Subsequently, the importance of data locality for scheduling threads motivated work stealing with data locality [1] wherein the data locality was discovered on the fly and maintained as the computation progressed. Their work also explored initial placement for scheduling and provided experimental results to show the usefulness of the approach; however, affinity was not always followed, the scope of the algorithm was limited to SMP environments and its time complexity was not analyzed. [4] analyzed time complexity ($O(T_1/P + T_\infty)$) for scheduling *general* parallel computations on SMP platform but does not consider space or message complexity bounds.

[6] considers work-stealing algorithms in a distributed-memory environment, with adaptive parallelism and fault-tolerance. Here task migration was entirely pull-based (via a randomized work stealing algorithm) hence it ignored affinity and also didn't provide any formal proof for the deadlock-freedom or resource utilization properties. The work in [2] described a *multi-place*(distributed) deployment for parallel computations for which initial placement based scheduling strategy is appropriate. A *multi-place* deployment has multiple places connected by an interconnection network where each *place* has multiple processors connected as in an SMP platform. It showed that online greedy scheduling of multithreaded computations may lead to physical deadlock in presence of bounded space and communication resources per place. Bounded resources(space or communication) can lead to cyclic dependency amongst the places which can lead to physical deadlock. [2] also provided a scheduling strategy based on initial placement and proved space bounds for physical deadlock free execution of *terminally strict* (section 2) computations by resorting to a degenerate mode called Doppelgänger mode. The computation did not respect affinity in this mode and no time or communication bounds were provided. Also, the aspect of load balancing was not addressed.

In this report, we propose novel affinity driven distributed scheduling algorithms and prove space, time and message bounds while guaranteeing deadlock free execution. The algorithms assume initial placement annotations on the given parallel computation with consideration of load balance *across* the places. The algorithms control the online expansion of the computation DAG using an efficient remote spawn and reject handling mechanism across places and randomized work stealing *within* a place for load balancing. The scheduling algorithm for bounded space uses *depth* based ordering for execution of activities to ensure deadlock free execution. These algorithms can be easily extended to variable number of processors per place and also to mapping multiple logical places in the program to the same physical place, provided the physical place has sufficient resources. To the best of our knowledge this is the first time affinity driven deadlock-free distributed scheduling algorithms have been designed and analyzed in a *multi-place* setup for both unconstrained and bounded space. Our main contributions are:

- We present an online affinity driven multi-place distributed scheduling algorithm for *strict* place-annotated multi-threaded computations under assumption of unconstrained space per place. We show that the lower bound on the expected execution time is $O(\max_k T_1^k/m + T_{\infty,n})$ and the upper bound is $O(\sum_k (T_1^k/m + T_\infty^k))$, where k is a variable that denotes places from 1 to n, m denotes the number of processors per place, T_1^k denotes the execution time on a single processor for place k, and $T_{\infty,n}$ denotes the execution time of the computation on n places with infinite processors on each place. We also derive probabilistic lower and upper bounds for time and communication complexity.

- For bounded space per place, we present a novel affinity driven distributed scheduling algorithm for terminally strict multi-place computations with provable deadlock free execution. We establish that the space bound per place is $O(m \cdot (D_{max} \cdot S_{max} + n \cdot S_{max} + S_1))$ and the lower bound on the time complexity is within $\log(D_{max})$ factor of the lower bound for the unconstrained space case, where, D_{max} is the maximum depth of the computation in terms of activities and S_{max} is the size of largest activation frame.

2 System and Computation Model

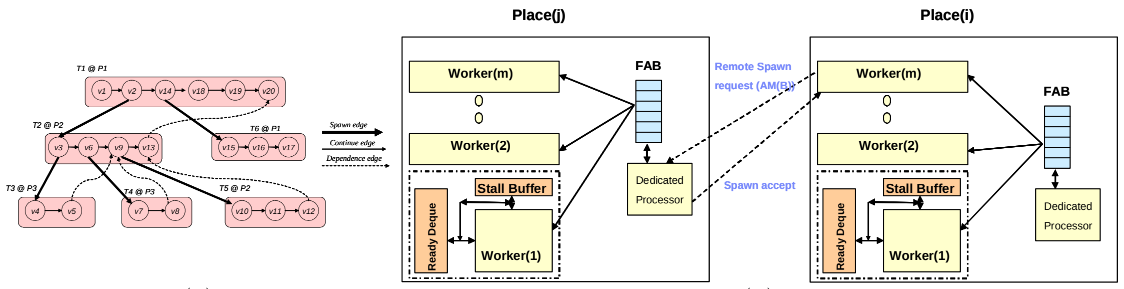
The system on which the *computation DAG* is scheduled is assumed to be cluster of *SMPs* connected by an *Active Message Network*. Each *SMP* is a group of processors with shared memory. Each SMP is also referred to as *place* in the report. Active Messages (*(AM)*[7] is a low-level lightweight RPC(remote procedure call) mechanism that supports unordered, reliable delivery of matched request/reply messages. We assume that there are n places and each place has m processors(also referred to as workers).

The parallel computation, to be dynamically scheduled on the system, is assumed to be specified by the programmer in languages such as X10 and Chapel.

[7]Active Messages defined by the AM-2: http://www.lrr.in.tum.de/ weissc/am.html

To describe our distributed scheduling algorithms, we assume that the parallel computation has a *DAG*(directed acyclic graph) structure and consists of nodes that represent basic operations like *and*, *or*, *not*, *add* and others. There are edges between the nodes in the computation DAG (Fig. 1(a)) that represent creation of new activities (*spawn* edge), sequential execution flow between nodes within a thread/activity (*continue* edge) and synchronization dependencies (*dependence* edge) between the nodes. In the report we refer to the parallel computation to be scheduled as the *computation DAG*. At a higher level the parallel computation can also be viewed as a computation tree of *activities*. Each *activity* is a *thread* (as in multi-threaded programs) of execution and consists of a set of nodes (basic operations). Each activity is assigned to a specific place (affinity as specified by the programmer). Hence, such a computation is called *multi-place* computation and DAG is referred to as *place-annotated* computation DAG (Fig. 1(a): v1..v20 denote nodes, T1..T6 denote activities and P1..P3 denote places).

The structure of dependencies between the nodes can vary depending on the input parallel computation. In *fully-strict* and *strict* computations the dependencies can go from a node to its immediate parent and to any of its ancestors in the computation DAG, respectively. In a *terminally strict* computation, introduced in [2] and shown in Fig. 1(a), the dependencies arise due to an activity waiting for the completion of its descendants. Every dependency edge, therefore, goes from the last instruction of an activity to one of its ancestor activities with the following restriction: In a subtree rooted at an activity called Γ_r, if there exists a dependence edge from any activity in the subtree to the root activity Γ_r, then there cannot exist any dependence edge from the activities in the subtree to the ancestors of Γ_r. A *terminally strict multi-place* computation is defined as a terminally strict computation where each activity has an affinity to a place.



(a) (b)

Figure 1: (a) Place-annotated Computation Dag. (b) Distributed Scheduling.

2.1 Useful Notations

The set of places is denoted by $P = \{P_1, \cdots, P_n\}$. The set of workers at place P_i, is denoted by $\{W_i^1, W_i^2..W_i^m\}$. S_1 denotes the space required by a single processor execution schedule. The size in bytes of the largest activation frame in the computation is denoted by S_{max}. If node u enables node v then we place an edge, referred as *enable* edge from u to v. The tree formed over all nodes with enable edges is referred to as *enabling tree* [4]. $depth(u)$ denotes the distance of node u from the root in the enabling tree. The root node is assumed to be at depth 0. $T_{\infty,n}$ denotes the execution time of the computation DAG over n places with infinite processors at each place. T_{∞}^{k} denotes the execution time for activities at place P_k using infinite processors. Note that, $T_{\infty,n} \leq \sum_{1 \leq k \leq n} T_{\infty}^{k}$. T_1^k denotes the minimum time taken by a single processor for the activities assigned to place k. D_{max} denotes the maximum depth of the computation tree in terms of number of activities. The *depth* of an activity is defined as the distance from the root activity in the computation tree.

3 Affinity Driven Distributed Scheduling in Unconstrained Space

Consider a *strict* place-annotated computation DAG. The distributed scheduling algorithm described below schedules activities with affinity at only their respective places. Within a place, work-stealing is enabled to allow load-balanced execution of the computation sub-graph associated with that the place. The computation DAG unfolds in an online fashion in a breadth-first manner across places when the affinity driven activities are pushed onto their respective remote places. Within a place, the online unfolding of the computation DAG happens in a depth-first manner to enable efficient space and time execution. Since sufficient space is guaranteed to exist at each place, physical deadlocks due to lack of space cannot happen in this algorithm.

3.1 Algorithm Design

Each place maintains *Fresh Activity Buffer* (*FAB*) which is managed by a dedicated processor(different from workers) at that place. Each worker at a place has a *Ready Deque* and *Stall Buffer* (refer Fig. 1(b)). The FAB at each place as well as the Ready Deque at each worker use concurrent deque implementation. An activity that has affinity for a remote place is pushed into the FAB at that place. An idle worker at a place will attempt to randomly steal work from other workers at the same place(*randomized work stealing*). Note that an activity which is pushed onto a place can move between workers at that place (due to work stealing) but can not move to another place and thus obeys affinity at all times. The distributed scheduling algorithm is given in Fig. 2.

3.2 Space Bound Analysis

The space computation obtains the total space requirement across all places (and processors) for the algorithm described in section 3.1 and given in Fig. 2.

At any step, an activity A at the r^{th} worker (at place i) W_i^r, may perform following actions:

1. **Spawn**:

   (a) A spawns activity B at place,P_j, $i \neq j$: A sends *AM(B)* (active message for B) to the remote place. Since, the remote place, P_j, is guaranteed to have memory for activity B, it is successfully inserted in the FAB at P_j and A continues execution (Fig. 1(b)).

   (b) A spawns B locally: B is successfully created and starts execution whereas A is pushed into the bottom of the Ready Deque.

2. **Terminates** (A terminates): The worker at place P_i, W_i^r, where A terminated, picks an activity from the bottom of the Ready Deque for execution. If none available in its Ready Deque, then it steals from the top of other workers' Ready Deque. Each failed attempt to steal from another worker's Ready Deque is followed by attempt to get the topmost activity from the FAB at that place. If there is no activity in the FAB then another victim worker is chosen from the same place.

3. **Stalls** (A stalls): An activity may stall due to dependencies in which case it is put in the stall buffer in a stalled state. Then same as *Terminates* (case 2) above.

4. **Enables** (A enables B): The termination of an activity A may enable a stalled activity B in which case the state of B changes to enabled and it is pushed onto the top of the Ready Deque.

Figure 2: Distributed Scheduling Algorithm

Let D be the computation dag representing the multi-threaded computation, and let the root of the computation be denoted by r. Let τ denote a node in the computation dag. Let *C-MHP*(τ)=$\{C_1, C_2, \ldots, C_k\}$ denote the sets of immediate children of τ such that

- the sets are disjoint
- two elements belonging to same set may happen in parallel.
- elements from different sets can never execute in parallel

The space required (denoted by $O(size(r))$) by the entire computation dag D for getting scheduled is of the order of maximum number of nodes(threads) that can execute in parallel. This is given by the following recursive equation:

$$size(r) = Max_{C \in C\text{-}MHP(r)}((|C|) + \sum_{c \in C} size(c)) \tag{3.1}$$

The notation $Max_{C \in C\text{-}MHP(\mathrm{r})}$ means maximum of the value computed for each of the elements in *C-MHP*(r).

3.3 Time Complexity Analysis

The time complexity of this affinity driven distributed scheduling algorithm in terms of number of *throws* and ready nodes in the computation is presented below. Each *throw* represents an attempt by a worker(*thief*) to steal an activity from another worker(*victim*) at the same place (intra-place work stealing).

Lemma 3.1 *Consider a strict place-annotated computation DAG with work per place, T_1^k, being executed by the affinity driven distributed scheduling algorithm presented in section 3.1. Then, the execution (finish) time for place,k, is $O(T_1^k/m + Q_r^k/m + Q_e^k/m)$, where Q_r^k denotes the number of throws when there is at least one ready node at place k and Q_e^k denotes the number of throws when there are no ready nodes at place k. The lower bound on the execution time of the full computation is $O(\max_k(T_1^k/m + Q_r^k))$ and the upper bound is $O(\sum_k(T_1^k/m + Q_r^k/m))$.*

Proof At any place, k, we collect tokens in three buckets: *work bucket*, *ready-node-throw bucket* and *null-node-throw bucket*. In the work bucket the tokens get collected when the processors at the place k execute the ready nodes. Thus, the total number of tokens collected in the work bucket is T_1^k. When, the place has some ready nodes and a processor at that place throws or attempts to steal ready nodes from the PrQ or another processor's deque then the tokens are added to the read-node-throw bucket. If there are no ready nodes at the place then the throws by processors at that place are accounted for by placing tokens in the null-node-throw bucket. The tokens collected in these three buckets account for all work done by the processors at the place till the finish time for the computation at that place. Thus, the finish time at the place k, is

$O(T_1^k/m + Q_r^k/m + Q_e^k/m)$. The finish time of the complete computation DAG is the maximum finish time over all places. So, the execution time for the computation is $\max_k O(T_1^k/m + Q_r^k/m + Q_e^k/m)$. We consider two extreme scenarios for Q_e^k that define the lower and upper bounds. For the lower bound, at any step of the execution, every place has some ready node, so there are no tokens placed in the null-node-throw bucket at any place. Hence, the execution time per place is $O(T_1^k/m + Q_r^k/m)$. The execution time for the full computation becomes $O(\max_k(T_1^k/m + Q_r^k/m))$. For the upper bound, there exists a place, say (w.l.o.g.) s, where the number of tokens in the null-node-throw buckets, Q_e^s, is equal to the sum of the total number of tokens in the work buckets of all other places and the tokens in the read-node-throw bucket over all other places. Thus, the finish time for this place, T_f^s, which is also the execution time for the computation is given by:

$$T_f^s = O(\sum_{1 \le k \le n} (T_1^k/m + Q_r^k/m)) \tag{3.2}$$

4 Time Complexity Analysis: Scheduling Algorithm with Unconstrained Space

We compute the bound on the number of tokens in the ready-node-throw bucket using potential function based analysis as given in [4]. Our unique contribution is in proving the lower and upper bounds of time complexity and message complexity for **multi-place** distributed scheduling algorithm presented in section 3.1 that involves **both** *intra-place work stealing* and *remote place affinity driven work pushing.*

Let there be a non-negative potential with ready nodes (each representing one instruction) in a computation dag. During the execution using the affinity driven distributed scheduling algorithm 3.1, the weight of a node u in the *enabling tree*, $w(u)$ is defined as $(T_{\infty,n} - \mathit{depth}(u))$, where $\mathit{depth}(u)$, is the depth of u in the enabling tree of the computation. For a ready node, u, we define $\phi_i(u)$, the potential of u at timestep i, as:

$$\phi_i(u) = 3^{2w(u)-1}, \text{if u is assigned;} \tag{4.1a}$$

$$= 3^{2w(u)}, \text{otherwise} \tag{4.1b}$$

All non-ready nodes have 0 potential. The potential at step i, ϕ_i, is the sum of the potential of each ready node at step i. When an execution begins, the only ready node is the root node with potential, $\phi(0) = 3^{2T_{\infty,n}-1}$. At the end the potential is 0 since there are no ready nodes. Let E_i denote the set of processes whose deque is empty at the beginning of step i, and let D_i denote the set of all other processes with non-empty deque. Let, F_i denote the set of all ready nodes present in the FABs of all places. The total potential can be partitioned into three parts as follows:

$$\phi_i = \phi_i(E_i) + \phi_i(D_i) + \phi_i(F_i) \tag{4.2}$$

where,

$$\phi_i(E_i) = \sum_{q \epsilon E_i} \phi_i(q) = \sum_{1 \le k \le n} \phi_i^k(E_i); \tag{4.3a}$$

$$\phi_i(D_i) = \sum_{q \epsilon D_i} \phi_i(q) = \sum_{1 \le k \le n} \phi_i^k(D_i); \tag{4.3b}$$

$$\phi_i(F_i) = \sum_{q \epsilon F_i} \phi_i(q) = \sum_{1 \le k \le n} \phi_i^k(F_i); \tag{4.3c}$$

where, $\phi_i^k()$ are respective potential components per place k. The potential at the place k, ϕ_i^k, is equal to the sum of the three components, i.e.

$$\phi_i^k = \phi_i^k(E_i) + \phi_i^k(D_i) + \phi_i^k(F_i) \tag{4.4}$$

Actions such as assignment of a node from deque to the processor for execution, stealing nodes from the top of victim's deque and execution of a node, lead to decrease of potential (refer Lemma 4.2). The idle processors at a place do work-stealing alternately between stealing from deque and stealing from the FAB. Thus, $2P$ throws in a round consist of P throws to other processor's deque and P throws to the FAB. We first analyze the case when randomized work-stealing takes place from another processor's deque using *balls and bins* game to compute the expected and probabilistic bound on the number of throws. For uniform and random throws in the balls and bins game it can be shown that one is able to get a constant fraction of the reward with constant probability (refer Lemma 4.3). The lemma below shows that whenever P or more throws occur for getting nodes from the top of the deques of other processors at the same place, the potential decreases by a constant fraction of $\phi_i(D_i)$ with a constant probability. For algorithm in section 3.1, $P = m$ (only intra-place work stealing).

Lemma 4.1 *Consider any round i and any later round j, such that at least P throws have taken place between round i (inclusive) and round j (exclusive), then, $Pr\{(\phi_i - \phi_j) \ge 1/4.\phi_i(D_i)\} > 1/4$*

For proof of this lemma refer [4]. There is an additional component of potential decrease which is due to pushing of ready nodes onto remote FABs. Let the potential decrease due to this transfer be $\phi_{i \to j}^k(out)$. The new probabilistic bound becomes:

$$Pr\{(\phi_i - \phi_j) \ge 1/4.\phi_i(D_i) + \phi_{i \to j}^k(out)\} > 1/4 \tag{4.5}$$

The throws that occur on the FAB at a place can be divided into two cases. In the first case, let the FAB have at least $P = m$ activities at the beginning of round i. Since, all m throws will be successful, we consider the tokens collected from such throws as work tokens and assign them to the work bucket of the respective processors. In the second case, in the beginning of round i, the FAB has less than m activities. Therefore, some of the m throws might be unsuccessful. Hence, from the perspective of place k, the potential $\phi_i^k(F_i)$ gets reduced to zero. The potential added at place k in $\phi_j^k(F_j)$ is due to ready nodes pushed from the deque of other places. Let this component be $\phi_{i-j}^k(in)$. The potential of the FAB at the beginning of round j is:

$$\phi_j^k(F_j) - \phi_i^k(F_i) = \phi_{i\to j}^k(in), \tag{4.6}$$

Furthermore, at each place the potential also drops by a constant factor of $\phi_i^k(E_i)$. If a process q in the set E_i^k does not have an assigned node, then $\phi_i(q) = 0$. If q has an assigned node u, then $\phi_i(q) = \phi_i(u)$ and when node u gets executed in round i then the potential drops by at least $5/9.\phi_i(u)$. Adding over each process q in E_i^k, we get:

$$\{\phi_i^k(E_i) - \phi_j^k(E_j)\} \geq 5/9.\phi_i^k(E_i). \tag{4.7}$$

Lemma 4.2 *The potential function satisfies the following properties*

1. *When node u is assigned to a process at step i, then the potential decreases by at least* $2/3\phi_i(u)$.
2. *When node u is executes at step i, then the potential decreases by at least* $5/9\phi_i(u)$ *at step i.*
3. *For any process, q in* D_i*, the topmost node u in the deque for q maintains the property that:* $\phi_i(u) \geq 3/4\phi_i(q)$
4. *If the topmost node u of a processor q is stolen by processor p at step i, then the potential at the end of step i decreases by at least* $1/2\phi_i(q)$ *due to assignment or execution of u.*

Lemma 4.3 Balls and Bins Game: *Suppose that at least P balls are thrown independently and uniformly at random into P bins, where for i = 1,2...P, bin i has weight* W_i. *The total weight* $W = \sum_{1\leq i\leq P} W_i$. *For each bin i, define a random variable,* X_i *as,*

$$\begin{aligned} X_i &= W_i, \textit{if some ball lands in bin } i \qquad &(4.8a)\\ &= 0, \textit{otherwise} &(4.8b)\end{aligned}$$

If $X = \sum_{1\leq i\leq P} X_i$*, then for any* β *in the range* $0 < \beta < 1$*, we have* $Pr\{X \geq \beta.W\} > 1 - 1/((1-\beta)e)$

We now establish the lower and upper bounds on time complexity.

Theorem 4.4 *Consider any strict place-annotated multi-threaded computation, being executed by the affinity driven multi-place distributed scheduling algorithm (section 3.1). Let the critical-path length for the computation be* T_∞. *The lower bound on the expected execution time is* $O(\max_k T_1^k/m + T_{\infty,n})$ *and the upper bound is* $O(\sum_k (T_1^k/m + T_\infty^k))$. *Moreover, for any* $\epsilon > 0$*, the execution time is* $O(\max_k T_1^k/m + T_{\infty,n} + \log(1/\epsilon))$ *with probability at least* $1 - \epsilon$.

Proof Lemma 3.1 provides the lower bound on the execution time in terms of number of throws. We shall prove that the expected number of throws per place is $O(T_\infty \cdot m)$, and that the number of throws per place is $O(T_\infty \cdot m + \log(1/\epsilon))$ with probability at least $1 - \epsilon$.

We analyze the number of ready-node-throws by breaking the execution into phases of $\theta(P = mn)$ throws ($O(m)$ throws per place). We show that with constant probability, a phase causes the potential to drop by a constant factor, and since we know that the potential starts at $\phi_0 = 3^{2T_{\infty,n}-1}$ and ends at zero, we can use this fact to analyze the number of phases. The first phase begins at step $t_1 = 1$ and ends at the first step, t'_1, such that at least P throws occur during the interval of steps $[t_1, t'_1]$. The second phase begins at step $t_2 = t'_1 + 1$, and so on.

Combining equations (4.5), (4.6) and (4.7) over all places, the components of the potential at the places corresponding to $\phi^k_{i\to j}(out)$ and $\phi^k_{i\to j}(in)$ cancel out. Using this and Lemma 4.1, we get that: $Pr\{(\phi_i - \phi_j) \geq 1/4.\phi_i\} > 1/4$.

We say that a phase is successful if it causes the potential to drop by at least a 1/4 fraction. A phase is successful with probability at least 1/4. Since the potential drops from $3^{2T_{\infty,n}-1}$ to 0 and takes integral values, the number of successful phases is at most $(2T_{\infty,n} - 1)\log_{4/3} 3 < 8T_{\infty,n}$. The expected number of phases needed to obtain $8T_{\infty,n}$ successful phases is at most $32T_{\infty,n}$. Since each phase contains $O(mn)$ ready-node throws, the expected number of ready-node-throws is $O(T_{\infty,n} \cdot m \cdot n)$ with $O(T_{\infty,n} \cdot m)$ throws per place. The high probability bound can be derived [4] using Chernoff's Inequality. We omit this for brevity.

Now, using Lemma 3.1, we get that the lower bound on the expected execution time for the affinity driven multi-place distributed scheduling algorithm is $O(\max_k T^k_1/m + T_{\infty,n})$.

For the upper bound, consider the execution of the subgraph of the computation at each place. The number of throws in the ready-node-throw bucket per place can be similarly bounded by $O(T_\infty k.m)$. Further, the place that finishes the execution in the end, can end up with number of tokens in the null-node-throw bucket equal to the tokens in work and read-node-throw buckets of other places. Hence, the finish time for this place, which is also the execution time of the full computation DAG is $O(\sum_k (T^k_1/m + T^k_\infty))$. The probabilistic upper bound can be similarly established using Chernoff bound.

The following theorem bounds the message complexity of the affinity driven work stealing algorithm described in section 3.1.

Theorem 4.5 *Consider the execution of a strict place-annotated computation DAG with critical path-length $T_{\infty,n}$ by the Affinity Driven Distributed Scheduling Algorithm in section 3.1. Then, the total number of bytes communicated across places is $O(I(S_{max} + n_d))$ and the lower bound on the total number of bytes communicated within a place has the expectation $O(m.n.T_{\infty,n}.S_{max}.n_d)$, where n_d is the maximum number of join edges from the descendants to a parent, S_{max} is the size in bytes of the largest activation frame in the computation and I is* *the number of remote spawns from one place to a remote place as specified in the place-annotations of the DAG. Moreover, for any $\epsilon > 0$, the probability is at least $(1-\epsilon)$ that the lower bound on the communication overhead incurred is $O(m.n.(T_\infty + \log(1/\epsilon)).n_d.S_{max})$.*

Proof First consider inter-place messages. Let the number of affinity driven pushes to remote places be $O(I)$, each of $O(S_{max})$ bytes. Further, there could be at most n_d dependencies from remote descendants to a parent, each of which involves communication of constant, $O(1)$, number of bytes. So, the total inter place communication is $O(I.(S_{max} + n_d))$. Since the randomized work stealing is within a place, the lower bound on the expected number of steal attempts per place is $O(m.T_\infty)$ with each steal attempt requiring $O(S_{max})$ bytes of communication within a place. Further, there can be communication when a child thread enables its parent and puts the parent into the child processors' deque. Since this can happen n_d times for each time the parent is stolen, the communication involved is at most $(n_d \cdot S_{max})$. So, the expected total intra-place communication per place is $O(m \cdot T_{\infty,n} \cdot S_{max} \cdot n_d)$. The probabilistic bound can be derived using Chernoff's inequality and is omitted for brevity. Similarly, expected and probabilistic upper bounds can be established for communication complexity within the places.

The computation is guaranteed to be deadlock-free using algorithm 3.1 if space given by equation (3.1) is available on the system. [2] shows deadlock-freedom for terminally strict computations when run on bounded resources by resorting to Doppelgänger mode. However, the Doppelgänger mode could take large time. In the next section, we present an algorithm that schedules activities such that deadlock freedom is guaranteed in presence of bounded resources while ensuring exact affinity mapping.

5 Affinity Driven Distributed Scheduling in Bounded Space

Due to limited space on real systems, the distributed scheduling algorithm has to limit online breadth first expansion of the computation DAG while minimizing the impact on execution time and simultaneously providing deadlock freedom guarantee. Due to bounded space constraints this distributed online scheduling algorithm has guaranteed deadlock free execution for *terminally strict* multi-place computations.

Due to space constraints at each place in the system, the activities can be stalled due to lack of space. The algorithm needs to keep track of space availability at each worker and place to ensure physical deadlock freedom. When an activity is stalled due to lack of space at a worker, it moves into *local-stalled* state. When an activity is stalled as it cannot be spawned onto a remote place, it moves into *remote-stalled* state. An activity that is stalled due to synchronization dependencies is moves into *depend-stalled* state.

Our distributed scheduling algorithm ensures ***distributed physical deadlock avoidance*** by using *depth based ordering* of computations for execution. Physical deadlock freedom ensures that the system always has space to guarantee the execution of a certain number of paths, that can vary during the execution of the computation DAG. If a place can locally expand an entire depth first path of an activity down to its leaf, then it is said to guarantee the *Assured Depth Expansion* property for that activity. The scheduling algorithm enforces that this property holds for work pushing as well as intra-place work stealing. A place that is pushing work, is allowed to do so only if the other place can guarantee the Assured Depth Expansion property on the work getting pushed. To provide good time and message bounds the distributed deadlock avoidance scheme is designed to have low communication cost while simultaneously exposing maximal concurrency inherent in the place-annotated computation DAG.

We assume that maximum depth of the computation tree (in terms of number of activities), D_{max}, can be estimated fairly accurately prior to the execution from the parameters used in the input parallel computation. D_{max} value is used in our distributed scheduling algorithm to ensure physical deadlock free execution. The assumption on knowledge of D_{max} prior to execution holds true for the kernels and large applications of the **Java Grande Benchmark suite** [8]. The D_{max} for kernels including *LUFact* (LU factorization), *Sparse* (Sparse Matrix multiplication), *SOR* (successive over relaxation for solving finite difference equations) can be exactly found from the dimension of input matrix and/or number of iterations. For kernels such as *Crypt* (International Data Encryption Algorithm) and *Series* (Fourier coefficient analysis) the D_{max} again is well defined from the input array size. The same holds for applications such as *Molecular Dynamics*, *Monte Carlo Simulation* and *3D Ray Tracer*. Also, for graph kernels in the **SSCA#2 benchmark** [9], D_{max} can be known by estimating $\triangle_g$ (diameter) of the input graph (e.g. $O(polylog(n))$ for R-MAT graphs, $O(\sqrt{n})$ for DIMACS graphs).

5.1 Distributed Data-Structures & Algorithm Design

Each worker has the following data-structures (Fig. 3(a)):

- *PrQ* and *StallBuffer*: *PrQ* is a priority queue that contains activities in enabled state and *local-stalled* state. The *StallBuffer* contains activities in *depend-stalled* and *remote-stalled* states. The total size of both these data-structures together is $O(D_{max} \cdot S_{max})$ bytes.

- *Ready Deque*: Also referred to as Deque, this contains activities in the current executing path on this worker. This has total space of $O(S_1)$ bytes.

- *AMRejectMap*: This is a one-to-one map from a place-id,say P_j, to the tuple [U, $AM(V)$, head, tail]. This tuple contains, $AM(V)$, the active

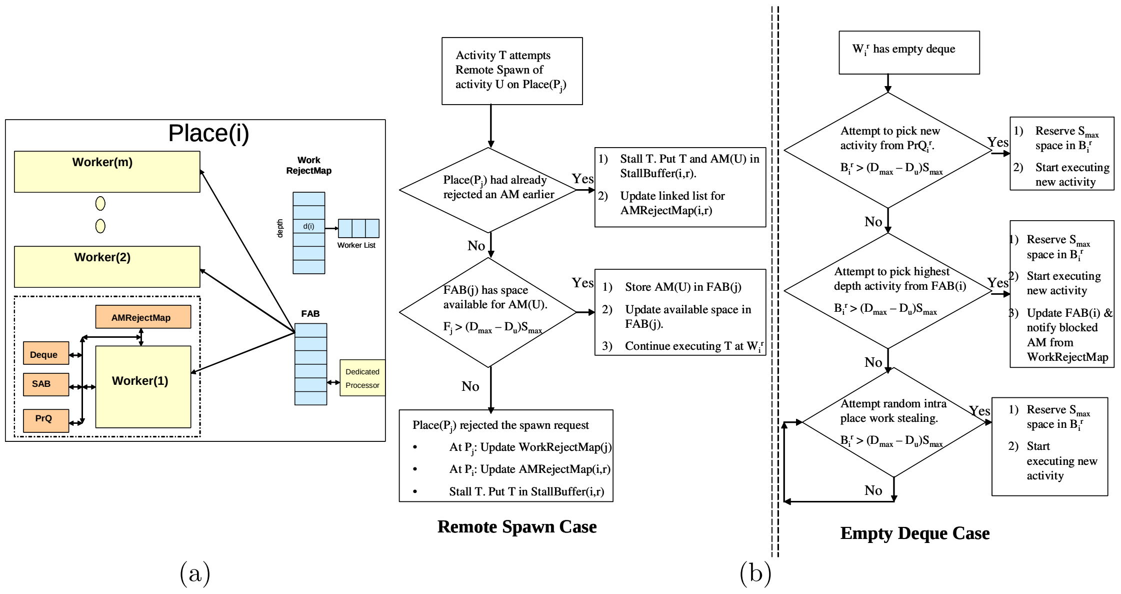
[8]http://www.epcc.ed.ac.uk/research/activities/java-grande
[9]http://www.highproductivity.org/SSCABmks.htm

message rejected in a remote-spawn attempt at place P_j; U, the activity stalled due to the rejected active message and head and tail of the linked list of activities in *remote-stalled* state due to lack of space on the place, P_j. This map occupies $O(n \cdot S_{max})$ space per worker.

Each place P_i, has the following data-structures (Fig. 3(a)):

- *FAB*: This is a concurrent priority queue that is managed by a dedicated processor (different from workers). It contains the fresh activities spawned by remote places onto this place. It occupies $O(D_{max} \cdot S_{max})$ bytes in space.
- *WorkRejectMap*: This is a one-to-many map from depth to list of workers. For each depth this map contains the list of workers whose spawns were rejected from this place. It occupies $O(m \cdot n + D_{max})$ space.

The priority queue (used for *PrQ* and FAB) uses the *depth* of an activity as the priority with higher depth denoting higher priority. The *depth* of an activity is defined as the distance from the root activity in the computation tree. The algorithm description uses the following notation. Let *AMRejectMap(i,r)*,



Place(i)
Worker(m)
Worker(2)
Worker(1)
AMRejectMap
Deque
SAB
PrQ
Work RejectMap
depth
d(i)
Worker List
FAB
Dedicated Processor
Activity T attempts Remote Spawn of activity U on Place(P_j)
Place(P_j) had already rejected an AM earlier
Yes
1) Stall T. Put T and AM(U) in StallBuffer(i,r).
2) Update linked list for AMRejectMap(i,r)
No
FAB(j) has space available for AM(U). F_j > (D_max – D_u)S_max
Yes
1) Store AM(U) in FAB(j)
2) Update available space in FAB(j).
3) Continue executing T at W_i^r
No
Place(P_j) rejected the spawn request
• At P_j: Update WorkRejectMap(j)
• At P_i: Update AMRejectMap(i,r)
• Stall T. Put T in StallBuffer(i,r)
Remote Spawn Case
W_i^r has empty deque
Attempt to pick new activity from PrQ_i^r. B_i^r > (D_max – D_u)S_max
Yes
1) Reserve S_max space in B_i^r
2) Start executing new activity
No
Attempt to pick highest depth activity from FAB(i) B_i^r > (D_max – D_u)S_max
Yes
1) Reserve S_max space in B_i^r
2) Start executing new activity
3) Update FAB(i) & notify blocked AM from WorkRejectMap
No
Attempt random intra place work stealing. B_i^r > (D_max – D_u)S_max
Yes
1) Reserve S_max space in B_i^r
2) Start executing new activity
No
Empty Deque Case
(a)
(b)

Figure 3: (a) Distributed Data Structures For Bounded Space Scheduling Algorithm. (b) Remote Spawn and Empty Deque Case In Bounded Space Scheduling Algorithm.

PrQ(i,r) and *StallBuffer(i,r)* denote the *AMRejectMap*, *PrQ* and *StallBuffer* respectively for worker W_i^r at place P_i. Let B_i^r denote the combined space for the *PrQ(i,r)* and *StallBuffer(i,r)*. Let *FAB(i)* and *WorkRejectMap(i)* denote the

FAB and *WorkRejectMap* respectively at place P_i. Let F_i denote the current space available in *FAB(i)*. Let *AM(T)* denote the active message for spawning the activity T. The activities in *remote-stalled* state are tracked using a linked list using activity IDs with the head and tail of the list available at the tuple corresponding to the place in map *AMRejectMap*.

Computation starts with root of the computation DAG which is at depth 1. The computation starts at a worker W_0^s, at the default place P_0. At any point of time a worker at a place, W_i^r, can either be executing an activity,T, or be idle. The detailed algorithm is presented in Fig. 4. An intuitive description of each of the phases is given below.

When T spawns a local activity U (**Local Spawn** case) there is guaranteed space to execute it and hence it is simply pushed to the bottom of the Deque(at that worker) and child activity U is taken up for execution. When T needs to attempt a remote spawn(**Remote Spawn** case, refer Fig. 3(b)) at place P_j, it first checks if there are already stalled activities in *AMRejectMap(i,r)*. If there is already a stalled activity, then T is added to the *StallBuffer(i,r)* and the link from the current tail in the tuple corresponding to P_j in *AMRejectMap(i,r)* is set to T. Also, the tail of the tuple is set to T. If there is no stalled activity in *AMRejectMap(i,r)* for place P_j, then the worker attempts a remote spawn at place P_j. At P_j check is performed by the dedicated processor for space availability in the *FAB(j)*. If it has enough space then the active message,$AM(U)$, is stored in the remote *FAB(j)*, the available space in *FAB(j)* is updated and T continues execution. If there isn't enough space then *AMRejectMap(i,r)* is updated accordingly and T is put in the *StallBuffer(i,r)*. When the worker W_i^r receives notification(**Receives Notification** case) of available space from place P_j, then it gets the tuple for P_j from *AMRejectMap(i,r)* and sends the active message and the head activity to P_j. At P_j, the *WorkRejectMap(j)* is updated. Also, W_i^r updates the tuple for P_j by updating the links for the linked list in that tuple. When currently executing activity T terminates (**Terminates** case), then the worker picks the bottommost activity to execute if one such exists. Else, the space reserved by T is released from B_i^r and the worker follows the same as the case for *Empty Deque*. When the *Deque* becomes empty (**Empty Deque** case, refer Fig. 3(b)) then the worker attempts to pick activity from its *PrQ*. If that does not succeed, then it tries to pick activity from *FAB(i)*. If that also fails then it looks for any entries in the map *WorkRejectMap* and sends notification to the appropriate worker whose activity it can execute. If this also fails then the worker tries to randomly steal an activity of appropriate depth from another worker at the same place. When an activity U gets enabled (**Activity Enabled** case) then it is moved from *StallBuffer* into the *PrQ*. If the activity was in *remote-stalled* state and it gets enabled due to notification from another place, then the actions as in the case *Receives Notification* are also performed. When an activity T stalls (**Activity Stalled** case) then its state is set to appropriate stalled state and it is removed from *Deque* and put in either *PrQ* or *StallBuffer* depending on whether it is in *local-stalled* state or not. The next bottommost activity U is picked from the *Deque*. If there is enough space to execute this activity then it is picked for execution else it is stalled.

At any time, a worker W_i^r takes the following actions. It might be executing an activity T(@depth D_t).

1. ***Local Spawn*: T spawns activity U locally**. T is pushed to the bottom of the *Deque*. U starts executing.
2. ***Remote Spawn*: T attempts remote spawn of U(@depth D_u) at a remote place P_j, $i \neq j$**
   // Refer Remote Spawn Case Flow Chart in Fig. 3(b)
   - If (*AMRejectMap(i,r)*[P_j] at W_i^r is non-null)
     (a) Let U be the value of tail in the tuple for P_j in *AMRejectMap(i,r)*.
     (b) Stall T. Put T and associated active msg for spawn into *StallBuffer*.
     (c) Update link of U to TaskID(T).
     (d) Update value of tail in the tuple for P_j in *AMRejectMap(i,r)*.
   - else If ($|F_j| > (D_{max} - D_u)S_{max}$) //corresponds to assured depth expansion
     (a) Store $AM(U)$ in FAB_j and update F_j, as, $F_j \leftarrow F_j - S_{max}$.
     (b) T continues execution.
   - else //P_j rejects the request and data-structures get updated as:
     (a) At place P_j, Update *WorkRejectMap(j)*: Insert pair $< D_u, W_i^r >$.
     (b) At place P_i, Update *AMRejectMap(i,r)*: Insert pair $< P_j, < AM(U), head(T), tail(null) >>$.
     (c) At place P_i, Activity T put into *StallBuffer(i,r)*.
3. ***Receives Notification*: W_i^r receives notification from place P_j on available space for spawn**
   (a) Get pair $< P_j, < R, AM(V), head(U), tail(S) >>$ from *AMRejectMap(i,r)*.
   (b) Send the tuple , $< AM(V), U >$, to Place P_j. Put R in PrQ_i^r.
   (c) Update head in the tuple for the pair with key as P_j as: $head = U \rightarrow$Next().
4. ***Termination*: T terminates**
   - if(*Deque(i,r)* is non-empty) then Pick the bottommost activity from *Deque(i,r)*
   - else //Let b be the amount of space reserved by activity T.
     (a) Release space. $B_i^r \leftarrow B_i^r + b$.
     (b) Same as case *Empty Deque*.
5. ***Empty Deque*: W_i^r has an empty deque**
   // Refer Empty Deque Case Flow Chart in Fig. 3(b)
6. ***Activity Enabled*: activity U gets enabled**
   (a) Set state of U to enabled.
   (b) Insert U in PrQ_i^r.
   (c) if(activity enabled was in state *remote-stalled*) then Perform other actions as in the case *Receives Notification*.
7. ***Activity Stalled*: T (@depth D_t) stalls**
   Let U(@depth D_u) be the next bottommost activity in *Deque(i,r)*.
   - State of T is changed to an appropriate stalled state. T is removed from *Deque*. If T is in *local-stalled* state (due to lack of space at worker) then it is moved into *PrQ* else it is moved into *StallBuffer*.
   - If(Enough space already reserved by T)
     (a) Update the space already reserved by T.
     (b) Execute U.
   - else if(Enough space available in B_i^r considering already reserved space by T)
     (a) Reserve further space. Update B_i^r.
     (b) Execute U.
   - else { Put U in stalled state.}

Figure 4: Multi-place Distributed Scheduling Algorithm

5.2 Case Study of Bounded Space Scheduling Algorithm using BFS on undirected graph

This section presents the trace of the bounded space distributed scheduling algorithm (Fig. 4), with space available as $O(m \cdot n \cdot D_{max} \cdot S_{max})$ per place. The scheduling algorithm is run on an undirected graph $G(V, E)$ (Fig. 5) with $|V| = 8$ and $|E| = 10$. The trace is partial for brevity but is representative of the way the execution will proceed using the bounded space distributed scheduling algorithm. The BFS algorithm (X10 pseudo-code) as a terminally strict computation is presented below.

```
initialization;
void BFS(RootNode, currNode)
begin
    // update distance to nbrNode
    finish for each nbrNode of currNode
        if local(nbrNode) then
            async(UD(currNode,nbrNode));
        end
        else
            async(UD(currNode,nbrNode)) at Place(place(nbrNode));
        end
    end
    // explore graph from nbrNodes
    finish for each nbrNode of currNode
        if local(nbrNode) then
            async(BFS(RootNode,nbrNode));
        end
        else
            async(BFS(RootNode,nbrNode)) at Place(place(nbrNode));
        end
    end
end
```

Algorithm 1: BFS Algorithm

BFS is carried out on simultaneously from two roots: v_1 and v_6. Note that centrality measures, like betweenness centrality (in network analysis) do BFS from all roots. We chose two roots for sake of simplicity. The system is assumed to have two places, $P1$ and $P2$ each with two processors, $m1$ and $m2$. The Program root, denoted by *Proot* initially starts execution at $P1 : m1$. The notation used in Fig. 6 is as follows. At each step S_i the state of the system changes The rows denotes the following:

- Row labeled *Deque* denotes state of the deque of each processor.
- Row labeled *ExecuteN* denotes the ready-node that is executing on each processor.
- Row labeled *Action* denotes the work performed by the executing ready-node. The work performed can be one of the following:
    - $ST(node, src)$ denotes steal work, *node*, from *src* which can be PrQ or another processor-deque.
    - $RS(node, placeId)$ denotes remote spawn of *node* on place *placeId*.
    - $LS(node)$ denotes local spawn.

  - $UD(bfsRootNodeId, v_j)$ denotes update of distance from bfsRootNode, $bfsRootNodeId$ to current graph-node, v_j.

- Row labeled PrQ denotes the contents of the PrQ at each place.
- Row labeled $Stall$ denotes nodes that get stalled per processor.

The nodes in the computation DAG are either labeled as $R_i v_j$ or $R_i v_j t_1$. The first notation refers to the bfs-thread with root R_i(i.e. graph-node v_i) and bfs-search proceeding further from graph-node v_j, while the second notation refers to the thread, t_1, for distance update of v_j for bfs root R_i(i.e. graph-node v_i).

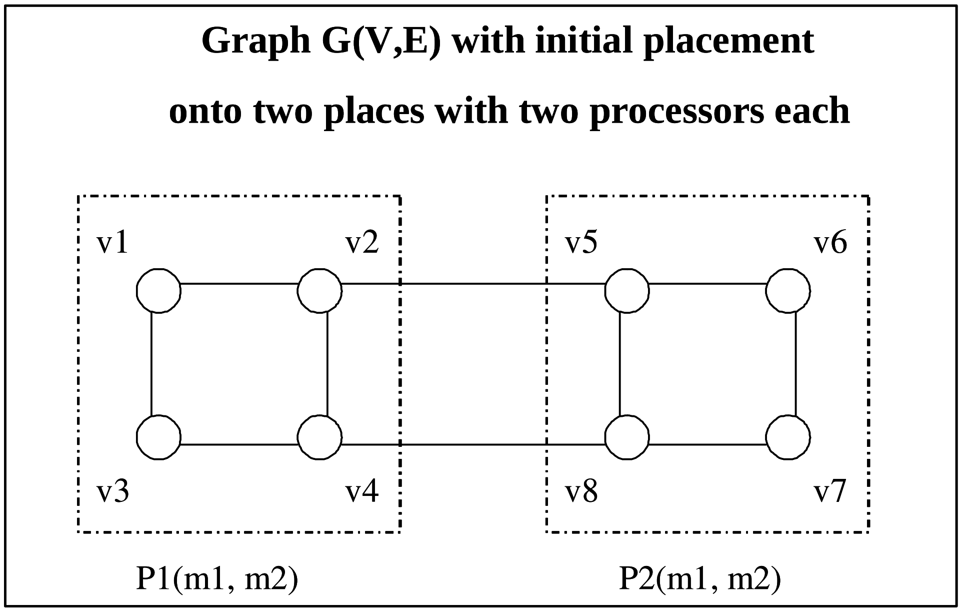

Graph G(V,E) with initial placement
onto two places with two processors each
v1
v2
v5
v6
v3
v4
v8
v7
P1(m1, m2)
P2(m1, m2)

Figure 5: Placement of BFS Graph On the System

5.3 Space Bound and Deadlock Freedom Proof

We prove in this section that the bounded space distributed scheduling algorithm (Fig. 4) executes in a deadlock free manner using limited space per place.

Lemma 5.1 *A place that accepts activity with depth d' has space to execute activities of depth greater than or equal to $d' + 1$.*

Proof: A place accepts depth d' activity if it has space greater $(D_{max} - d')$. The algorithm adopts a reservation policy which ensures that activities already executing have reserved space that they may need for stalled activities. The space required to execute activity of depth greater or equal to $d' + 1$ is obviously less, and hence, the place can execute it.

S#		Place(1)	Place(1)	Place(2)	Place(2)	S#		Place(1)	Place(1)	Place(2)	Place(2)
		Proc(m1)	Proc(m2)	Proc(m1)	Proc(m2)			Proc(m1)	Proc(m2)	Proc(m1)	Proc(m2)
S1	Execute N	Proot				S6	Deque			R6v6	
	Action	LS(R1v1)	ST(PrQ)	ST(m2)	ST(m1)		Execute N		R1v1	R6v5t1	
S2	Deque	Proot					Action	ST(PrQ)	LS(R1v3t1)	UD(R6,v5)	ST(PrQ)
	Execute N	R1v1				S7	Deque		R1v1		
	Action	LS(R1v2t1)	ST(Proot, m1)	ST(PrQ)	ST(PrQ)		Execute N		R1v3t1	R6v6	
S3	Deque	R1v1					Action	ST(m2)	UD(R1,v3)	LS(R6v7t1)	ST(m1)
	Execute N	R1v2t1	Proot			S8	Deque			R6v6	
	Action	UD(R1,v2)	RS(R6v6, P2)	ST(m2)	ST(m1)		Execute N	R1v1		R6v7t1	
S4	PrQ			R6v6			Action	LS(R1v2)	ST(PrQ)	UD(R6,v7)	ST(PrQ)
	Execute N	R1v1				S9	Deque	R1v1			
	Action	LS(R1v4t1)	ST(PrQ)	ST(R6v6, PrQ)	ST(PrQ)		Execute N	R1v2		R6v6	
S5	Deque	R1v1					Action	LS(R1v4t1)	ST(m1)	LS(R6v5)	ST(m1)
	Execute N	R1v4t1		R6v6		S10	Deque	R1v2		R6v6	
	Action	UD(R1,v4)	ST(m1)	LS(R6v5t1)	ST(m1)		Execute N	R1v4t1	R1v1	R6v5	
							Action	UD(R1->v4)	LS(R1v3)	LS(R6v7t1)	ST(PrQ)

Figure 6: Bounded Space Algorithm Trace For BFS with Two Roots

Lemma 5.2 *If a place rejects an activity of depth less than or equal to d', it implies that either an activity of depth greater or equal to d' will exist as perhaps a stalled activity or the workers when become idle shall release space so that d' depth activity can be executed.*

Proof: A place rejects a request of activity spawn of depth d' if does not have space atleast $(D_{max} - d')$. This is only possible if the workers that are executing an activity have reserved spaces such that the least required space is not available. Eventually the activities being executed either stall or terminate and workers may release some space. In case there is no released space, then there has to exist atleast one activity of depth greater or equal to d' because otherwise $D_{max} - d'$ worth of space would have been released.

Lemma 5.3 *An activity of depth $d' < D_{max}$ that terminates such that it is the last activity in the worker's deque shall enable an activity of depth $d' - 1$ at remote place if it cannot pick from its PrQ.*

Proof: When an activity of depth d' terminates, it releases $D_{max} - d'$ amount of space. Therefore, there is space available for creation of activity with depth d' which in turn enables its parent to proceed if it cannot pick anything from PrQ.

Lemma 5.4 *There is always space to execute activities at depth D_{max}.*

Proof: The space required to execute activities at D_{max} is 0 because it is the leaf activity and can always be executed to completion. Therefore, leaf activities get consumed from PrQ as soon as any worker gets idle. Space required in PrQ is atleast 1 to accept activities at depth D_{max}. When it is picked from PrQ, it makes a space available thereby paving way for enabling activities with depth less than D_{max}.

Lemma 5.5 *A path in the computation tree is guaranteed to execute.*

Proof: Let the max depth activity that a place P_1 is executing be d_1. Then the place is guaranteed to execute/accept an activity of d_2 depth such that $d_2 > d_1$ by Lemma 5.1. Therefore, this activity of depth d_1 if wants to create a child locally can do so without any trouble. Suppose, that it wants to create child at remote place P_2 and it rejects. By lemma 5.2, P_2 is executing an activity of depth atleast $d_1 + 1$. This activity of depth $d_1 + 1$ would have been created through some path in computation tree. Extending this argument, it can be seen that there exists a path across places that belongs to the computation tree such that it is a complete path. (A path from Root to Leaf in fully expanded computation tree is called a complete path.)

Theorem 5.6 *(Assured Leaf Execution) The scheduling maintains* assured leaf execution *property during computation. Assured leaf execution ensures that each node in computation tree becomes a leaf and gets executed.*

Proof We prove it by induction on depth of an activity in the computation tree.
Base case (depth of an activity is D_{max}):
By lemma 5.5, a path to a leaf is guaranteed. An activity at depth D_{max} is always a leaf and thus, an activity that occurs at D_{max} can always get executed at any place by lemma 5.4.
Induction Hypothesis: Let us assume that an activity at depth d is assured to become a leaf and get executed.
Induction Step: We have to show that an activity of depth $d-1$ is assured to become a leaf and get executed. An activity at $d-1$ may be a leaf already in the computation tree because it does not spawn any children in which case we just have to show that it does get scheduled. By induction hypothesis, we know that any activity of depth d shall get executed and thus by lemma 5.3 we can be sure that activity at $d-1$ shall get enabled and brought to PrQ where it will become highest priority (because activities of d are guaranteed to execute) and get scheduled. An activity at depth $d-1$ can spawn a local or remote child which is at depth d. By induction hypothesis, any such child will get executed. This implies the activity will get eventually get rid of the dependency on the children and become a leaf node in the computation tree. By the same arguments presented above, it can be shown that such an activity also gets scheduled and either it executes to create another set of child dependencies or it terminates. In the case it creates children, same argument holds and thus eventually it shall terminate because we are dealing with bounded computations.

Theorem 5.7 *A terminally strict computation scheduled using algorithm 4 uses* $O(m \cdot (D_{max} \cdot S_{max} + n \cdot S_{max} + S_1))$ *bytes as space per place.*

Proof The *PrQ*, *StallBuffer*, *AMRejectMap* and deque per worker (processor) take total of $O(m \cdot (D_{max} \cdot S_{max} + n \cdot S_{max} + S_1))$ bytes per place. The *WorkRejectMap* and *FAB* take total $O(m \cdot n + D_{max})$ and $O(D_{max} \cdot S_{max})$ space per place (section 5.1). The scheduling strategy adopts a space conservation policy to ensure deadlock free execution in bounded space. The basic aim of this strategy is to ensure that only as much breadth of a tree is explored as can be accommodated in the available space assuming each path can go to the maximum depth of D_{max}. It starts with the initial condition where available space is atleast $D_{max} \cdot S_{max}$ per worker per place. Any activity that gets scheduled on a worker reserves the space for the possible stalled activities that it can generate. The reserved space is released when an activity terminates or stalls. The amount of released space is typically of an order of the maximum possible depth of the subtree that the terminated/stalled activity would have had. This guarantees us lemma 5.1. No activity can be scheduled on a worker if it cannot reserve the space for the possible stalled activities that it can generate at that place. A place that enables a remote activity stalled because of space does so only after ensuring that appropriate amount of space is present for the activity that shall be created. Similarly, when a worker steals it will ensure that it has enough space to accommodate the activities that would get created as a result of execution of stolen activity. Whenever an activity of depth D_{max} is picked from PrQ it releases space, thus ensuring lemma 5.4. From the algorithm, it can be seen that every reservation and release is such that the total space requirement at a place does not exceed what was available initially. Hence, the total space per place used is $O(m \cdot (D_{max} \cdot S_{max} + n \cdot S_{max} + S_1))$.

5.4 Time and Message Bounds

The scheduling algorithm in Fig. 4 guarantees deadlock free execution with at least $O(D_{max}.S_{max})$ space per place. However, the available space per place is m times this space. Hence, in reality many concurrent paths can execute in parallel in the system. The space-time trade-off analysis of this algorithm is left for future work. The difference between this work stealing algorithm, Fig. 4 and the one in section 3.1 is that this algorithm works under bounded space and uses concurrent PrQ for FAB per place and also per worker. This leads to two consequences on the work done. First, the spawn of a child onto a remote place might get rejected in which case the node stalls at the source place but the additional work done to take care of rejects for remote spawns is constant i.e. $O(1)$. The other consequence is that a throw onto the PrQ from a local worker involves more than just a typical constant time deque operation. If we consider non-blocking concurrent priority queue implementation [9] for the PrQ then it takes $O(log(D_{max}))$ time to perform the concurrent insertion and deletion of an activity from the PrQ. Since, the number of successful throws at a place can be $O(T_1^k)$, so the additive time complexity from these PrQ operations is $O((T_1^k/m).\log(D_{max}))$. This leads us to the following theorem on the lower bound for the time complexity of the bounded space scheduling algorithm, Fig. 4.

Theorem 5.8 *Consider any multi-threaded computation with work per place denoted by T_1^k, being executed by the multi-place affinity driven distributed scheduling algorithm, Fig. 4. The lower bound on the expected execution time is $O(\max_k(T_1^k/m) \cdot \log(D_{max}) + T_{\infty,n})$. Moreover, for any $\epsilon > 0$, the lower bound on the execution time is $O(\max_k(T_1^k/m) \cdot \log(D_{max}) + T_{\infty,n} + \log(1/\epsilon))$, with probability at least $(1 - \epsilon)$.*

The analysis of upper bound on time complexity involves modeling resource driven wait time and has been left for future work. The inter-place message complexity is same as theorem 4.5 as there is constant amount of work for handling rejected remote spawns and notification of space availability.

6 Related Work Comparison

[2] extends work stealing framework for terminally strict X10 computations and establishes deadlock free scheduling for SMP deployments. It proves deadlock free execution with bounded resources on uniprocessor cluster deployments while using Doppelgänger mode of execution. However, it neither considers

work stealing in this framework, nor, does it provide performance bounds. The Doppelgänger mode of execution can lead to arbitrary high costs in general. We consider affinity driven scheduling over an SMP cluster deployment using Active Message network. We further include intra-place and inter-place work stealing and prove space and performance bounds with deadlock free guarantee.

[1] considers nested-parallel computations on multiprocessor HSMSs (hardware-controlled shared memory systems) and proves upper-bounds on the number of cache-misses and execution time. It also presents a locality guided work stealing algorithm that leads to costly synchronization for each thread/activity. However, activities may not get executed at the processor for which they have affinity. We consider affinity driven scheduling in a multi-place setup and provide performance bounds under bounded space while guaranteeing deadlock free execution.

[4] provides performance bounds of a non-blocking work stealing algorithm, in a multiprogrammed SMP environment, for general multi-threaded computations under various kernel schedules using potential function technique. This approach however does not consider locality guided scheduling. We consider affinity driven multi-place work stealing algorithms, for applications running in dedicated mode (stand alone), with deadlock freedom guarantees under bounded resources and leverage the potential function technique for performance analysis.

[8] introduces *work-dealing* technique that attempts to achieve "locality oriented" load distribution on small-scale SMPs. It has a low overhead mechanism for dealing out work to processors in a global balanced way without costly compare-and-swap operations. We assume that the programmer has provided place annotations in the program in a manner that leads to optimal performance considering load-balancing. In our approach, the activities with affinity for a place are guaranteed to execute on that place while guaranteeing deadlock freedom.

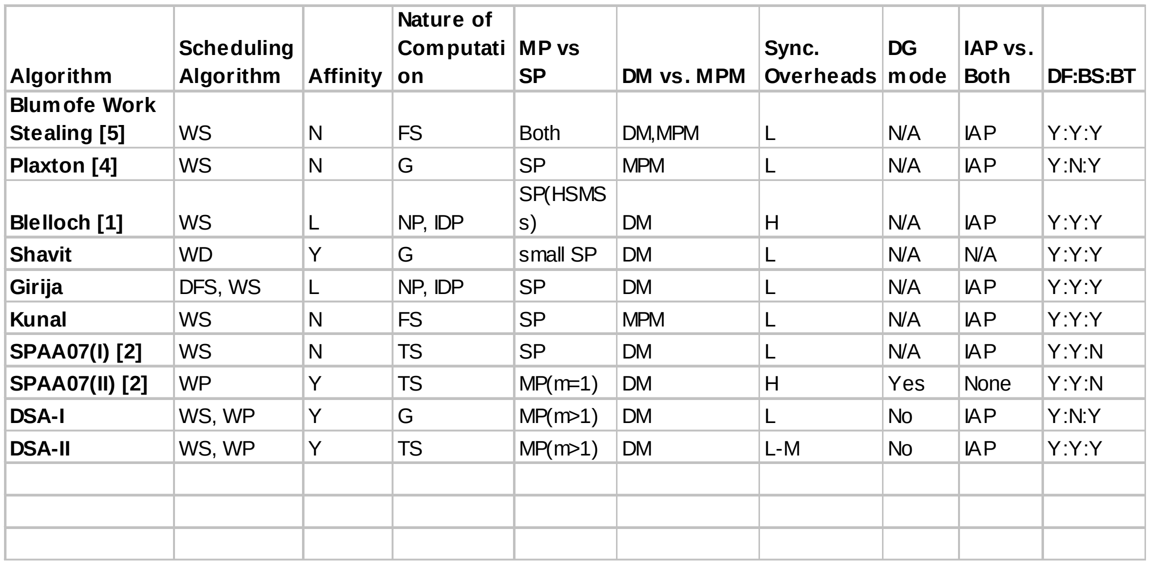
[10] presents a space-efficient scheduling algorithm for shared memory machines that combines the low scheduling overheads and good locality of work stealing with the low space requirements of depth-first schedulers. For locality it uses the heuristic of scheduling threads that are close in the computation DAG onto the same processor. We consider a multi-place setup and assume affinities in the place-annotate computation have been specified by the programmer.

[3] studies two-level adaptive multi-processor scheduling in an multi-programmed environment. It presents a randomized work-stealing thread scheduler for fork-join multithreaded jobs that provides continual parallelism feedback to the job scheduler in the form of requests for processors and uses *trim analysis* to obtain performance bounds. However, this work does not consider locality guided scheduling. We assume dedicated mode of execution but we believe our work can be extended to multiprogrammed mode also. We will explore this as part of future work.

Fig. 7 presents the comparison between different scheduling approaches. Our scheduling approaches are DSA I(unconstrained space) and DSA II(bounded space). The notation used is as follows.

- Column, *Scheduling Algorithm*, has values: WS(Work Stealing), WD(Work Dealing), DFS(Depth First Search) and WP(Work Pushing).
- Column, *Affinity Driven*, has values: Y(Yes), N(No) and L(limited extent).
- Column, *Nature Of Computation*, has values: FS(fully-strict), G(general), NP(nested parallel), IDP(iterative data parallel) and TS(terminally strict).
- Column, *MP vs SP*, denotes multi-place (MP) or single place(SP) algorithm setup.
- Column, *DM vs MPM*, denotes dedicated mode(DM) or multi-programmed mode(MPM) environment.
- Column, *Sync. Overheads*, has values L(low), M(medium) and H(high).
- Column, *DG mode*, denotes whether Doppelgänger mode is used in multi-place setup.
- Column, *IAP vs. Both*, denotes whether intra-place stealing(IAP) is only supported or both(Both) inter-place and intra-place stealing are supported.

The last column denotes whether deadlock freedom, space bound and time bound are presented in the respective scheduling approaches.



Algorithm	Scheduling Algorithm	Affinity	Nature of Computation	MP vs SP	DM vs. MPM	Sync. Overheads	DG mode	IAP vs. Both	DF:BS:BT
Blumofe Work Stealing [5]	WS	N	FS	Both	DM,MPM	L	N/A	IAP	Y:Y:Y
Plaxton [4]	WS	N	G	SP	MPM	L	N/A	IAP	Y:N:Y
Blelloch [1]	WS	L	NP, IDP	SP(HSMSs)	DM	H	N/A	IAP	Y:Y:Y
Shavit	WD	Y	G	small SP	DM	L	N/A	N/A	Y:Y:Y
Girija	DFS, WS	L	NP, IDP	SP	DM	L	N/A	IAP	Y:Y:Y
Kunal	WS	N	FS	SP	MPM	L	N/A	IAP	Y:Y:Y
SPAA07(I) [2]	WS	N	TS	SP	DM	L	N/A	IAP	Y:Y:N
SPAA07(II) [2]	WP	Y	TS	MP(m=1)	DM	H	Yes	None	Y:Y:N
DSA-I	WS, WP	Y	G	MP(m>1)	DM	L	No	IAP	Y:N:Y
DSA-II	WS, WP	Y	TS	MP(m>1)	DM	L-M	No	IAP	Y:Y:Y

Figure 7: Comparison Of Scheduling Approaches

7 Conclusions & Future Work

We have presented the design of novel distributed scheduling algorithms with provable deadlock freedom for both bounded and unconstrained space and space, time and message complexity analysis. This is first such work for distributed scheduling of parallel computations. In future we plan to look into space-time tradeoffs, markov-chain based modeling and multi-core implementations.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA*, pages 1 – 12, New York, NY, USA, December 2000.

[2] S. Agarwal, R.Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yellick. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA*, pages 229 – 240, San Diego, CA, USA, December 2007.

[3] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive work stealing with parallelism feedback. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 112 – 120, San Jose, California, USA, 2007.

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pages 119 – 129, Puerto Vallarta, Mexico, 1998.

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

[6] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX Annual Technical Conference*, Anaheim, California, 1997.

[7] 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*.

[8] Danny Hendler and Nir Shavit. Work Dealing. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 164 – 172, Winnipeg, Manitoba, Canada, 2002.

[9] M. Herlihy. Wait-Free Synchronization. *ACM Toplas*, 11(1):124 – 149, Jan. 1991.

[10] Girija Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 83 – 95, Saint Malo, France, 1999.