Deadlock Free Affinity Driven Distributed Scheduling Algorithm

Abstract

With the advent of many-core architectures efficient scheduling of parallel computations for higher productivity and performance has become very important. In order to realize scalable performance (reaching limits of exascale), it is necessary to have efficient scheduling of large parallel computations on huge number of cores/processors. The scheduler needs to execute in a pure distributed and online fashion, should follow affinity inherent in the computation and must have low time and message complexity. Further, it should also avoid physical deadlocks due to bounded resources including memory per core. Simultaneous consideration of these factors makes affinity driven distributed scheduling particularly challenging.

In this paper, we address this challenge by using distributed deadlock avoidance strategy under bounded space to ensure deadlock freedom, a low time and message complexity mechanism for ensuring affinity and randomized work-stealing for load balancing. This paper presents a novel affinity driven online distributed scheduling algorithm for multi-place (cluster of Symmetric Multiprocessors) parallel computations. If the input application has no logical deadlocks due to control, data or synchronization dependencies then, our scheduling algorithm guarantees deadlock free execution. We prove that our algorithm provides deadlock-free scheduling under bounded space per place. Using experimental results on well-known benchmarks, we demonstrate the superior performance and scalability of our algorithm as compared to randomized work stealing strategy. Our algorithm demonstrates efficient space-time trade-offs which highlight its robustness to multiple architectures. To the best of our knowledge, this is the first time an affinity driven deadlock-free distributed scheduling algorithm has been designed and analyzed in a multi-place setup with bounded space.

Distributed Scheduling; Multithreaded Computation; Algorithm; Work Stealing

1. Introduction

The exascale computing roadmap has highlighted efficient locality oriented scheduling in runtime systems as one of the most important challenges ("Concurrency and Locality" Challenge [9]). Massively parallel many core architectures have *NUMA* characteristics in memory behavior, with a large gap between the local and the remote memory latency. Unless efficiently exploited, this is detrimental to scalable performance. Languages such as X10 [8], Chapel [7] and Fortress [3] are based on partitioned global address space (PGAS [13]) paradigm. They have been designed and implemented as part of DARPA HPCS program [1] for higher productivity and performance on many-core massively parallel platforms. These languages have in-built support for initial placement of threads (also referred as activities) and data structures in the parallel program. Therefore, locality comes implicitly with the program. The run-time systems of these languages need to provide efficient algorithmic scheduling of parallel computations with medium to fine grained parallelism. Further, Petascale Analytics has become a very important area for both academia and industry. It involves handling petascale data and running compute and memory intensive algorithms on a large cluster. Here, moving data can be very costly hence affinity has to be obeyed and the thread has to be scheduled where the data resides.

For handling large parallel computations, the scheduling algorithm (in the run-time system) should be designed to work in a *distributed* fashion. This is also imperative to get scalable performance on many core architectures. Further, the execution of the parallel computation happens in the form of a dynamically unfolding execution graph. It is difficult for the compiler to always correctly predict the structure of this graph and hence perform correct scheduling and optimizations, especially for data-dependent computations. Therefore, in order to schedule generic parallel computations and also to exploit runtime execution and data access patterns, the scheduling should happen in an *online* fashion. Moreover, in order to mitigate the communication overheads in scheduling and the parallel computation, it is essential to follow *affinity* inherent in the computation. Simultaneous consideration of these factors along with low time and message complexity, makes distributed scheduling a very challenging problem.

Specifically, we address the following affinity driven distributed scheduling problem:

Given:

- An input computation DAG (Fig. 1) that represents a parallel multi-threaded computation with fine to medium grained parallelism. The DAG is defined as follows:
  - Each node in the DAG is a basic operation such as and/or/add etc. and is annotated with a *place* identifier which denotes where that node should be executed.
  - Each edge in the DAG represents one of the following: (i) spawn of a new thread or, (ii) sequential flow of execution or, (iii) synchronization dependency between two nodes.
  - The DAG is a *terminally strict* parallel computation DAG (synchronization dependency edge represents an activity waiting for the completion of a descendant activity, details in section 3).
- A cluster of *n* SMPs (refer Fig. 2) as the target architecture on which to schedule the computation DAG. Each SMP [2] also referred as *place* has fixed number(*m*) of processors and memory. The space per place is bounded as defined by the target architecture. The cluster of SMPs is referred as the *multi-place* setup.

[1] www.highproductivity.org/

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

Determine: An online schedule for the nodes of the computation DAG in a distributed fashion that ensures the following:

- Exact mapping of nodes onto *places* as specified in the input DAG.
- Physical deadlock free execution.
- Low time and message complexity for execution.
- Execute within the bounded space per place

In this paper, we present the design of a novel affinity driven, online, distributed scheduling algorithm with low time and message complexity while guaranteeing deadlock free execution. The algorithm assumes initial placement annotations on the given parallel computation with consideration of load balance *across* the places. The algorithm controls the online expansion of the computation DAG based on available space. The scheduling algorithm carefully manages space for execution in a distributed fashion using *computation depth upper-bound* based ordering of threads. This *distributed deadlock avoidance* strategy ensures deadlock free execution of the parallel computation and we prove this formally in this paper. Further, our algorithm employs an efficient remote spawn and reject handling mechanism across places for ensuring affinity. Randomized work stealing *within* a place helps in load balancing. Experimental results on well-known benchmarks (Heat, Molecular Dynamics, Conjugate Gradient) demonstrate the superior performance of our algorithm over Cilk [5] as well as space-time tradeoffs in distributed scheduling.

Our algorithm can be easily extended to variable number of processors per place and also to mapping of multiple logical places in the program to the same physical place, provided the physical place has sufficient resources. There are profiling tools available to help users visualize the data affinity of their programs for general parallel applications. The thread data affinity can be also be obtained using profiling and dynamic instrumentation tools such as Pin [3]. Using this affinity information, the programmer can easily specify the affinity with a fair degree of accuracy. Hence, our assumption that the programmer specifies the affinity (placement annotations), is a valid assumption in a practical scenario. For bounded space arguments, we consider only stack space for sake for clear explanation in this paper. Our algorithm and arguments can be easily extended to heap space. We leave this for future work. In this paper, we consider in-memory execution only as swapping to disk (to get more space) will lead to performance degradation and hence for deadlock free execution under bounded space we consider physical memory only (and not disk space).

Our main contributions are:

- We present a novel affinity driven, online, distributed scheduling algorithm for bounded space per place. This algorithm is designed for terminally strict multi-place parallel computations and ensures physical deadlock free execution using a distributed deadlock avoidance strategy.
- We present the space bound and deadlock freedom proof for this algorithm. Experimental results on well-known benchmarks (Heat, Molecular Dynamics, Conjugate Gradient) demonstrate superior performance over the simple work stealing based scheduling strategy [5]. Our algorithm demonstrates efficient space-time trade-offs for distributed scheduling.

2. Related Work

With the advent of multi-core / many-core architectures and cloud computing paradigm, distributed scheduling algorithms for parallel computations have become very important [11], [12], [10]. Scheduling of dynamically created tasks for shared memory multiprocessors 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* computations (synchronization dependency edges go from a thread to only its immediate parent thread, section 3) 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 the execution time on infinite processors. We consider locality oriented scheduling in distributed enviroments and hence are more general than Cilk. 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. This 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 only SMP environments and its time complexity was not analyzed. [4] analyzed the time complexity ($O(T_1/P + T_\infty)$) for scheduling *general* parallel computations on SMP platforms but does not consider locality oriented scheduling. We consider distributed scheduling problem across multiple places (cluster of SMPs), with bounded space per place, while ensuring affinity.

[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 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 (with example) that online greedy scheduling of multi-threaded computations may lead to physical deadlock in presence of bounded (stack) space and communication resources per place. As an example, consider a simple parallel computation, running on two places, as a sequence of spawns with the whole computation tree like a binary tree of finite depth (stack space per place is equivalent to this depth). Here, each spawn generates two new activites with affinity of both for the other place. As this computation (with *ping-pong spawns*) gets scheduled, the space available on each place might exhaust as the number of activity stack frames that get expanded might be expontential relative to the depth of the tree and hence the stack space will overflow. This wiill prevent computation tree from expanding further leading to a deadlock. This *physical* deadlock needs to be handled carefully. [2] suggest a way to handle this deadlock by violating the affinity which can lead to performance degradation. Also no time or communication bounds were provided. Also, the aspect of load balancing was not addressed even within a place. We ensure affinity along with intra-place load balancing in a multi-place setup. We also provide deadlock freedom proof and the space bound of our algorithm.

3. 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* (Fig. 2). Each *SMP* is a group of processors with shared memory. Each SMP is also referred to as *place* in the paper. Active

[3] www.pintool.org

Messages (*(AM)*[4] 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. To describe our distributed scheduling algorithm, 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 so forth. There are edges between the nodes in the computation DAG (Fig. 1) that represent creation of new activities (*spawn* edge), sequential execution flow between the nodes within a thread/activity (*continue* edge) and synchronization dependencies (*dependence* edge) between the nodes. In the paper 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: $v1..v20$ denote nodes, $T1..T6$ denote activities and $P1..P3$ denote places).

Based on the structure of dependencies between the nodes in the computation DAG, there can be following types of parallel computations:

1. ***Fully-strict computation***: Dependencies are only between the nodes of a thread and the nodes of its immediate parent thread.
2. ***Strict computation***: Dependencies are only between the nodes of a thread and the nodes of any of its ancestor threads.
3. ***Terminally strict computation***: (Fig. 1). 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.

The following notations are used in the paper:

- $P = \{P_1, \cdots, P_n\}$ denote the set of places.
- $\{W_i^1, W_i^2..W_i^m\}$ denote the set of workers at place P_i.
- S_1 denotes the space required by a single processor execution schedule.
- S_{max} denotes the size in bytes of the largest activation frame in the computation.
- D_{max} denotes the maximum depth of the computation tree in terms of number of activities.
- $T_{\infty,n}$ 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 assigned to place P_k using infinite processors. Note that, $T_{\infty,n} \leq \max_{1 \leq k \leq n} T_\infty^k$.
- T_1^k denotes the time taken by a single processor for the activities assigned to place k.

[4] Active Messages defined by the AM-2: http://now.cs.berkeley.edu/AM/active_messages.html

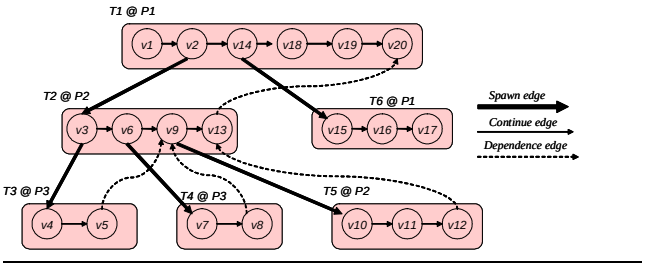

Figure 1. Place-annotated Computation Dag

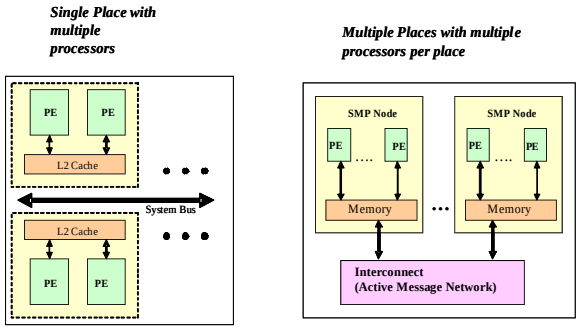

Figure 2. Multiple Places: Cluster of SMPs

4. Distributed Scheduling Algorithm

Consider a *terminally strict* place-annotated computation DAG. Our distributed scheduling algorithm 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.

Each place maintains a *Fresh Activity Buffer* (*FAB*) which is managed by a *dedicated processor* (which is a regular core/processor other than *workers* for a distributed environment) at that place. An activity that has affinity for a remote place is pushed (referred as *work-pushing*) into the FAB at that place. If the remote place FAB has sufficient space then it accepts the remote activity and it gets inserted in the remote FAB. Else, it rejects the remote spawn and the information about the rejected activity and worker is stored in both the source worker and the destination place (refer Fig. 3). The worker which initiated the remote spawn re-attempts this spawn only when sufficient space is guaranteed in the remote FAB. This helps in limiting the number of messages for a successful remote spawn.

For load balancing within a place, an idle worker at a place will attempt to randomly steal work from other workers at the same place (*randomized work stealing*). When a worker attempts to steal an activity from the *victim* worker, then it first checks whether it has sufficient space to execute that activity using the space it has available. The space required to execute an activity at depth d_i, is dependent on the remaining depth ($D_{max} - d_i$) with respect to the maximum depth (D_{max}) of the computation tree. The steal attempt is successful and the activity is stolen, if the *theif* worker has this space demanded by the stolen activity, else the *thief* makes another steal attempt. 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.

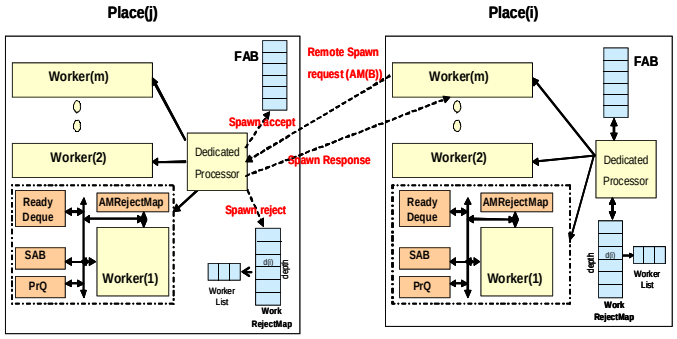

Figure 3. Distributed Data Structures & Spawn/Reject Mechanism Across Places

Due to limited space on real systems, this 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. To achieve this, our algorithm uses a novel ***distributed deadlock avoidance*** scheme. Due to space constraints at each place in the system, the activities can be stalled due to lack of space. The algorithm keeps track of stack space available on the system and that required by activities for execution (heap space is not considered for simplicity of discussion). The space required by an activity, u, is bounded by the maximum stack space needed for its execution, i.e. $((D_{max} - D_u) \cdot S_{max})$, where, D_{max} is the maximum activity depth in the computation tree , D_u is the depth of u in the computation tree, S_{max} is the size of the largest activation frame in the computation. The algorithm follows *depth based ordering* of computations for execution by allowing the activities with higher depth on a path to execute to completion before the activities with lower depth on the same path. This happens in a distributed fashion. Both during work-pushing and intra-place work stealing, each place and worker checks for availability of stack space for execution of the activity, respectively. Due to depth based ordering, only bounded number of paths in the computation tree are expanded at any point of time. This bound is based on the available space in the system. Using this distributed deadlock avoidance scheme, 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.

An activity can be in any of the multiple *stalled* states (*local-stalled* or *remote*-stalled or *depend*-stalled) during the execution of the scheduling algorithm. 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, it moves into *depend-stalled* state.

We assume that the 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 the knowledge of D_{max} prior to execution holds true for the kernels and large applications of the **Java Grande Benchmark suite** [5]. The D_{max} for kernels including *LUFact* (LU factorization), *Sparse* (Sparse Matrix multiplication), *SOR* (successive over relaxation for solving finite difference equations) can be found exactly from the dimension of the input matrix and/or the 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** [6], 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 [7] graphs).

4.1 Distributed Data-Structures & Algorithm Design

The distributed data structures used in the scheduling algorithm are described below. Each worker at a place, has the following data-structures (Fig. 3):

- ***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 has the following components:
  - $AM(V)$: active message rejected in a remote-spawn attempt at place P_j
  - U: activity stalled due to the rejected active message
  - $head$ 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):

- ***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 per place.
- ***WorkRejectMap***: This is a one-to-many map from the computation 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 (*computation*) *depth* of an activity is defined as the distance from the root activity in the computation tree.

Consider the following notations.

- $P = \{P_1, \cdots, P_n\}$ denotes the set of places.
- $\{W_i^1, W_i^2..W_i^m\}$ denote the set of workers at place P_i.
- S_1 denotes the space required by a single processor execution schedule.
- S_{max} denotes the size in bytes of the largest activation frame in the computation.
- D_{max} denotes the maximum depth of the computation tree in terms of number of activities.

Further, let *AMRejectMap(i,r)*, *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

[5] http://www.epcc.ed.ac.uk/research/activities/java-grande

[6] http://www.highproductivity.org/SSCABmks.htm

[7] http://dimacs.rutgers.edu/Challenges/index.html

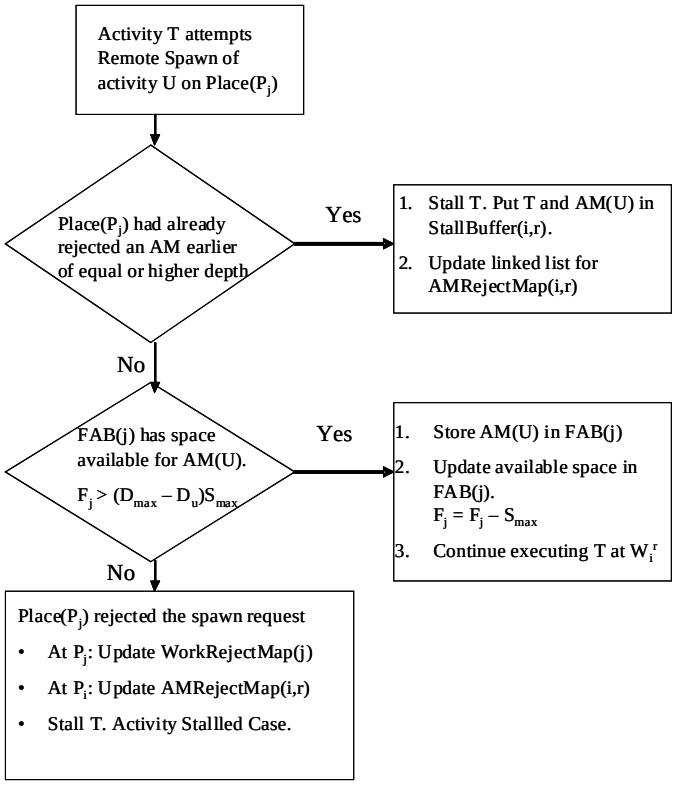

Figure 4. Remote Spawn Case in Distributed Scheduling Algorithm

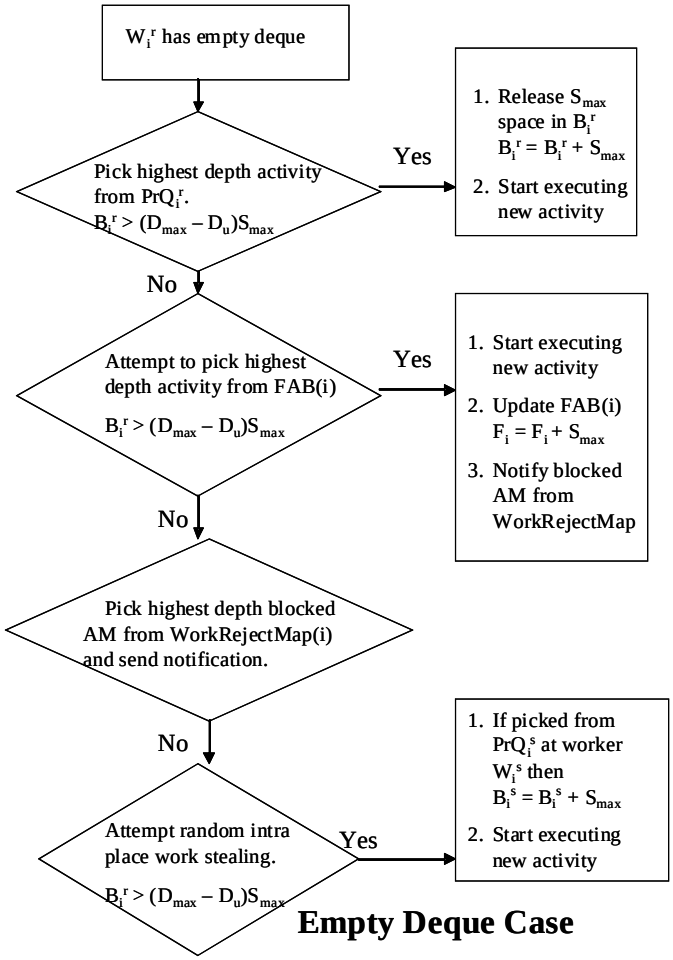

Figure 5. Empty Deque Case in Distributed Scheduling Algorithm

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 with 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 (depth 1) of the computation DAG at a worker W_0^s, at the default place P_0. At any point of time a worker, W_i^r, can either be executing an activity,T, or be idle. The detailed algorithm is presented in Fig. 6. The different cases of the algorithm are described here. When T needs to attempt a remote spawn (**Remote Spawn** case, refer Fig. 4) 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. The remote-stalled activity is enabled and put in *PrQ(i,r)* (**Activity Enabled** case). 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. 5) then the worker attempts to pick the available activity of maximum depth from its *PrQ*. If that does not succeed, then it tries to pick the available activity of maximum depth 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 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 also stalled.

4.2 Space Bound and Deadlock Freedom Proof

Since we consider stack space for execution in the space constraint, the depth of an activity in the computation tree is used in lemmas/proofs below. An activity at depth d requires less than $((D_{max} - d) \cdot S_{max})$ amount of stack space for execution since it can generate a maximum of $(D_{max} - d)$ stalled activities along

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 in Fig. 4

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 *AM-RejectMap(i,r)*.

   (b) Send the tuple , [$AM(V)$, U], to Place P_j.

   (c) Activity Enabled case for R. //Put R in PrQ_i^r.

   (d) Update head in the tuple for the pair with key as P_j as: $head$ = U →Next().

4. ***Termination*: T terminates**

   - if(*Deque(i,r)* is non-empty) then Pick the bottommost activity from *Deque(i,r)*
   - else { Same as case *Empty Deque*. }

5. ***Empty Deque*: W_i^r has an empty deque** // Refer Empty Deque Case in Fig. 5

6. ***Activity Enabled*: activity U gets enabled**

   (a) Set state of U to enabled. //no update for space

   (b) Insert U in PrQ_i^r.

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*.
     Update B_i^r. $B_i^r \leftarrow B_i^r - S_{max}$.
   - if($B_i^r > ((D_{max} - D_u) \cdot S_{max})$) { Execute U. }
   - else { Activity Stalled case for U.}

Figure 6. Multi-place Distributed Scheduling Algorithm

one execution path and each stack frame is bounded by S_{max} bytes. During the algorithm, this stack space ($((D_{max} - d) \cdot S_{max})$) is checked before picking the activity for execution (*Empty Deque Case*) or placing a remote active message in the FAB (*Remote Spawn* case). S_{max} space is reserved in the FAB when a remote active message is accepted and S_{max} space is released from the FAB when that active message is picked up by an idle worker for execution. S_{max} space is taken away from B_i^r when an activity gets stalled (*Activity Stalled* case), while S_{max} is added to B_i^r when that activity is picked up for execution (*Empty Deque* case).

LEMMA 4.1. *A place or a worker that accepts activity with depth d' has space to execute activities of depth greater than or equal to $(d' + 1)$.*

Proof At any point of time, a place or a worker accepts an activity of depth d' only if it has space greater than $(D_{max} - d') \cdot S_{max}$. This holds true in the *Remote Spawn*, *Empty Deque* and *Activity Stalled* cases of the algorithm (Fig. 6). The algorithm adopts this reservation policy which ensures that activities already executing have reserved space that they may need for stalled activities. The space required to execute an activity of depth greater or equal to $(d' + 1)$ is obviously less, and hence, the place can execute it.

LEMMA 4.2. *There is always space to execute activities at depth D_{max}.*

Proof The space required to execute activities at D_{max} is at most S_{max} because it is the leaf activity. Such activities do not depend on other activities and will not spawn any child activities so they will not generate any stalled activities. Hence, they require a maximum of S_{max} amount of space. Therefore, leaf activities get consumed from the PrQ as soon as its worker gets idle. The leaf activities also get pulled from the *FAB* and get executed by the worker, that has empty deque and cannot execute activities from its *PrQ* due to lack of activities or space.

LEMMA 4.3. *At any point of time (before the termination of complete computation tree execution) at least one path in the computation tree is guaranteed to execute.*

Proof : We use the depth based ordering property (valid during scheduling) in this 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 4.1. Therefore, this activity of depth d_1 if it wants to create a child locally (*Local Spawn* case) can do so without any trouble (lemma holds true). Else, suppose that it wants to create a child at a remote place P_2 and that place rejects (*Remote Spawn* and *Activity Stalled* case). Now, there are two cases. In the **first** case, P_2 has an active executing path, possibly not have reached depth d_1, but that is not stalled (lemma holds true). In the **second** case, P_2 is either executing an activity (at a worker at that place) of depth at least $d_1 + 1$ (lemma holds true) or has such an activity in stalled state. If this stalled state is depend-stalled state then an activity of depth even higher depth is executing at this or another place (lemma holds true). If this stalled state is local-stalled state, then there must be another activity of higher depth executing at that worker (lemma holds true). However, if the stalled state is remote-stalled state then we apply the same argument to the remote place on which this activity is waiting and we can see a monotonically increasing depth of activities in this resource dependency chain. Following this chain we will eventually hit an executing path due to cases discussed here or we have reached a leaf in the computation tree which can execute without dependencies (Lemma 4.2). Hence, it can be seen that there exists a path across places that belongs to the computation tree such that it is actively executing. Hence, at each instant of time there exists a path that is guaranteed to execute in the system. In fact, there can be multiple paths that are executing at any instant of time and this depends on the available space in the system and the computation tree.

THEOREM 4.4. *Assured Leaf Execution: The scheduling maintains* assured leaf execution *property during computation which ensures that each node in computation tree becomes a leaf and gets executed.*

Proof Proof by induction on the depth of an activity in the computation tree.
Base case (depth of an activity is D_{max}): By lemma 4.3, a path to a leaf is guaranteed. An activity at depth D_{max} is always a leaf and

has no dependencies on other activities. Thus, an activity that occurs at D_{max} will always get executed (by lemma 4.2). *Induction Hypothesis*: Let us assume that all activities at depth d and higher are assured to become leaves and get executed. *Induction Step*: We have to show that all activities of depth $(d-1)$ are also assured to become leaves and get executed. By induction hypothesis, the activities with depth d and higher have terminated. As in the *Termination* case, if there are remaining activities in the Deque then (they are at depth $(d-1)$) they become leaves and are picked up for execution. Otherwise, if the Deque becomes empty (*Empty Deque* case), the highest depth activities are picked for execution both from the *PrQ* and the *FAB*. Therefore, the activities at depth $(d-1)$ start execution. Further, the dependencies in the computation tree are from descendants to ancestors (terminally-strict computation). Therefore, when activities of depth d or higher finish execution, the activities at depth $(d-1)$, in depend-stalled or remote-stalled state, definitely become leaves and get enabled. Hence, they are put into the *PrQ* at the respective workers (*Activity Enabled* case). If the activity, at depth $(d-1)$, was in remote-stalled state, the blocked active message is sent to the remote place(*Receives Notification* case) for the spawn of child activity at depth d. By induction hypothesis, all activities at depth d have terminated so this has already happened earlier. Upon termination of d depth activity, assume the Deque is not empty and there are activities in *PrQ* of depth $(d-1)$. These activities wait till the current executing path in the Deque terminates. Then, these activities which have become leaves get picked up for execution (since they have the highest depth and have the highest priority in the *PrQ*). Hence, all activities at depth $(d-1)$ are also guaranteed to become leaves and get executed. Thus, our scheduling algorithm guarantees physical deadlock free execution.

THEOREM 4.5. *Space Bound: A terminally strict computation scheduled using algorithm in Fig. 6 takes $O(m \cdot (D_{max} \cdot S_{max} + n \cdot S_{max} + S_1))$ 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 4.1). 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. No activity (with depth D_u) can be scheduled on a worker if it cannot reserve the space for the possible stalled activities $(((D_{max} - D_u) \cdot S_{max}))$ that it can generate at that place (*Remote Spawn*, *Empty Deque* cases). 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 (*Activity Enabled* and *Receives Notification* cases). Similarly, when a worker steals it will ensure that it has enough space $(((D_{max} - D_u) \cdot S_{max}))$ to accommodate the stalled activities that would get created as a result of execution of stolen activity, u (*Empty Deque* case). When an activity gets stalled (*Activity Stalled* case) it reserves S_{max} space from B_i^r and when it is picked up for execution (*Empty Deque* case) it release this space, S_{max} from B_i^r. So, the space B_i^r suffices during execution. Similarly, for the *FAB*, S_{max} space is reserved when an active message is placed and S_{max} space is release when that active message is picked for execution by an idle worker (*Empty Deque* case). Thus, the FAB space requirement does not exceed during execution. The check on the FAB space for remote spawn $((((D_{max} - D_u) \cdot S_{max}))$ ensures depth-based ordering of activities across places and hence helps in deadlock free execution. 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. Results & Analysis

We implemented our distributed scheduling algorithm (*ADS*) and the pure Cilk style work stealing based scheduler (*CWS*) using pthreads (NPTL) API. The code was compiled using gcc version (4.1.2) with options *-O2* and *-m64*. In these experiments, we used multi-core Intel architecture and hence *dedicated* processor was not used for any place. Hence, both Cilk style scheduling and our distributed scheduling used exactly the same number of cores. Using well known benchmarks the performance of *ADS* was compared with *CWS* and also with the original Cilk [8] scheduler (referred as *CORG* in this section). These benchmarks are:

- **Heat**: Jacobi over-relaxation that simulates heat propagation on a two dimensional grid for a number of steps. In the Heat benchmark, two grids (arrays) are maintained. In each step of the algorithm, the values from the first 2D array are used to update the second 2D array, which was updated in the previous step. For our scheduling algorithm (*ADS*), the grid is partitioned uniformly across the available cores. The D_{max} for this benchmark is $\log(numCols/leafmaxcol)$ where *numCols* represents the number of columns in the input two-dimensional grid and *leafmaxcol* represents the number of columns to be processed by a single thread.
- **Molecular Dynamics** (MD): This is the classical Molecular Dynamics simulation, using the Velocity Verlet time integration scheme. The simulation was carried for a certain number of iterations. In each iteration the force between each pair of particle is computed and the position of each particle is updated based on the total force on that particle.
- **Conjugate Gradient** (NPB [9] benchmark): Conjugate Gradient (CG) approximates the largest eigenvalue of a sparse, symmetric, positive definite matrix using inverse iteration. The matrix is generated by summing outer products of sparse vectors, with a fixed number of nonzero elements in each generating vector. The benchmark computes a given number of eigenvalue estimates, referred to as outer iterations, using 25 iterations of the CG method to solve the linear system in each outer iteration.

The performance comparison was done on the Intel multi-core NUMA architecture. Although our algorithm is designed for cluster environment, we performed experiments on Intel multi-core architecture for ease of implementation and even here our affinity driven scheduling delivers performance gains against cilk type scheduling. The Intel multi-core platform has 16 cores (2.93 GHz, intel Xeon 5570, Nehalem architecture) with $8MB$ $L3$ cache per chip and around 64GB memory. Intel Xeon 5570 has NUMA characteristics even though it exposes SMP style programming. We did not use hyper-threading on Intel architecture. Fig. 7 compares the performance for the Heat benchmark (number of cores: 8, number of iterations = 1000, leafmaxcol = 32). Both *ADS* and *CWS* demonstrate data scalability. For matrix sizes varying from $32K * 4K$ to $64K * 4K$, the time for *CWS* is worse than the time for *ADS* by an average of 31%. Further, with increasing matrix size, *ADS* shows increasing gains as compared to *CWS*. This demonstrates that our affinity driven distributed scheduling algorithm is not only much better than Cilk style scheduling but its relative performance improves with increasing data, which is a necessary characteristic

[8] http://supertech.csail.mit.edu/cilk/

[9] http://www.nas.nasa.gov/NPB/Software

for large scale parallel applications involving GBs to TBs of data (and higher). We also compared our performance to original cilk, *CORG*. Here, *ADS* delivers upto around $2\times$ performance improvement as compared to *CORG* on the same number of cores. On larger distributed memory architectures, the performance gain of our algorithm will be much higher due to the locality advantage of our algorithm over *CORG* and *CWS*. This is because while cilk style scheduling will result in data movement, our algorithm will ensure that threads are spawned where the data is located and hence will avoid large data movement. This is a significant advantage delivered by our algorithm. Even algorithms such as [1] do not obey locality at all times and hence do not perform well for distributed execution.

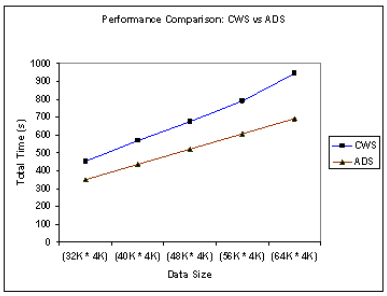

Figure 7. Performance Comparison: CWS vs ADS

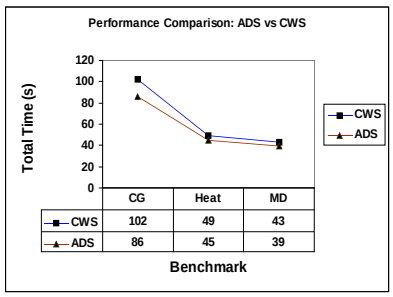

Figure 8. ADS vs CWS: Performance

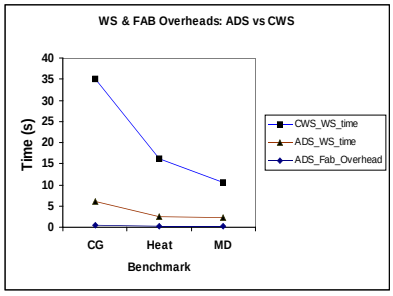

Figure 9. ADS vs CWS: Overheads

5.1 Detailed Performance Analysis

In this section, we analyze the performance gains obtained by our *ADS* algorithm vs. the Cilk style scheduling (*CWS*) algorithm. We also study the variation in the *FAB* and *Ready Deque* space and the variation in work-stealing time and *FAB* stealing time, with changes in number of places and processors (workers) per place. Space-time trade-offs in distributed scheduling are also analyzed.

Fig. 8 demonstrates the gain in performance of *ADS* vs *CWS* with 16 cores. *ADS* used 2 places and 8 processors (workers) per place. For CG, Class C matrix is chosen with parameters: *NA* = $75K$, *Non-Zero* = $15M$, *Outer iterations* = 75, *SHIFT* = 110. For Heat, the parameters values chosen are: matrix size = $2K * 4K$, number of iterations = 2000 and leafmaxcol = 32. For MD, $4K$ molecules are involved in force computation and distance update for 200 iterations. While CG has maximum gain of around 16%, Heat shows gain of 8.1% and MD shows gain of 9.3%. Fig. 9 demonstrates the overheads due to work stealing and FAB stealing in *ADS* and *CWS*. *ADS* has lower work stealing overhead because the work stealing happens only within a place. For CG, work steal time for *ADS* ($6s$) is $5.8\times$ better than *CWS* ($35s$). For Heat and MD, *ADS* work steal time is $6.4\times$ and $4.6\times$ better respectively, as compared to *CWS*. *ADS* has *FAB* overheads but this time is very small, around 6% to 13% of the corresponding work steal time. *CWS* has higher work stealing overhead because the work stealing happens from any place to any other place. Hence, the NUMA delays add up to give a larger work steal time. This demonstrates the superior execution efficiency of our algorithm over *CWS*.

Fig. 10 plots the variation of the work stealing time, the FAB stealing time and the total time with changing configurations of a multi-place setup, for MD benchmark. With constant total number of cores (workers) $= 16$, the configurations, in the format *(number of places * number of processors per place)*, chosen are: (a) $(2*8)$, (b) $(4*4)$, (c) $(8*2)$, and (d) $(16*1)$. As the number of places increase from 2 to 8, the work steal time increases from $2.1s$ to $4.5s$ as the average number of work steal attempts increases from $120K$ to $300K$. For 16 places, the work steal time falls to 0, as here there is only a single processor per place, so work stealing does not happen. The FAB steal time, however, increases monotonically from $0.2s$ for 2 places, to $19s$ for 16 places. In the $(16*1)$ configuration, the processor at a place gets activities to execute, only through remote push onto its place. Hence, the FAB steal time at the place becomes high, as the number of FAB attempts ($120M$ average) is very large, while the successful FAB attempts are very low (1100 average). With increase in the number of places from 2 to 16, the total time increases from $38.6s$ to $86s$, due to increase in the work stealing and/or the FAB steal overheads.

Fig. 11 plots the work stealing time and *FAB* stealing time variation with changing multi-place configurations for the CG benchmark. In this case, the work steal time remains similar for both $(2*8)$ and $(8*2)$ configurations, at around $5.1s$, and then falls to 0 for the $(16*1)$ configuration. The *FAB* time also initially remains constant for 2 to 8 places, but then jumps to $31s$ for $(16*1)$ configuration. The total time increases to $156s$ at 16 places, from $87s$ at 2 places.

Fig. 12 plots the work stealing time and *FAB* stealing time variation with changing multi-place configurations for the Heat benchmark. The variation of work stealing time, FAB stealing time and total time follow the pattern as in the case of MD.

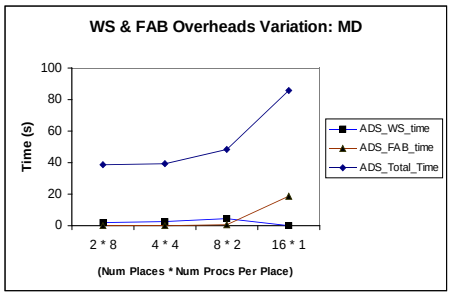

Figure 10. Variation of Overheads With Multi-Place Configurations

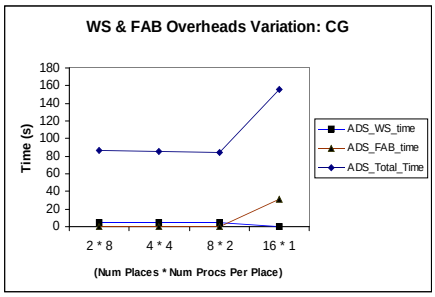

Figure 11. Variation of Overheads With Multi-Place Configurations

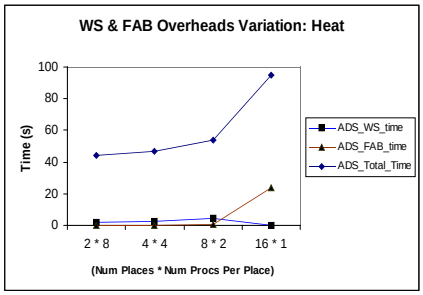

Figure 12. Variation of Overheads With Multi-Place Configurations

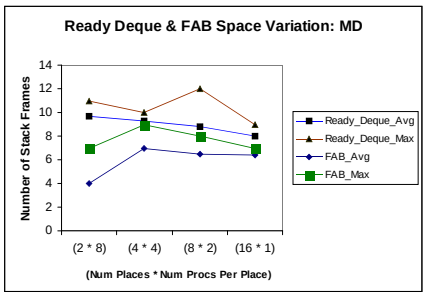

Figure 13. Variation of Space Utilization with Multi-Place Configurations

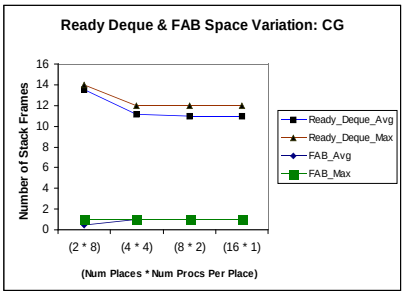

Figure 14. Variation of Space Utilization with Multi-Place Configurations

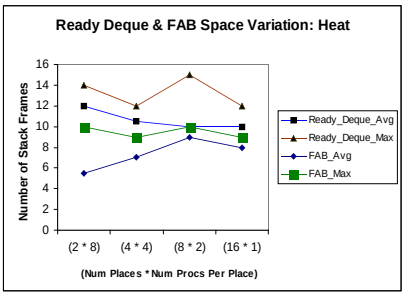

Figure 15. Variation of Space Utilization with Multi-Place Configurations

Fig. 13 gives the variation of the *Ready Deque* average space and maximum space consumption across all processors and *FAB* average space and maximum space consumption across places, with changing configurations of the multi-place setup. As the number of places increase from 2 to 16, the *FAB* average space increase from 4 to 7 stack frames first, and, then decreases to 6.4 stack frames. The maximum *FAB* space usage increases from 7 to 9 stack frames but then returns back to 7 stack frames. The average *Ready Deque* space consumption increases from 11 stack frames to 12 stack frames but returns back to 9 stack frames for 16 places, while the average *Ready Deque* monotonically decreases from 9.69 to 8 stack frames. The D_{max} for this benchmark setup is 12 stack frames, which leads to 75% maximum *FAB* utilization and 100% *Ready Deque* utilization.

Fig. 14 gives the variation of *FAB* space and *Ready Deque* space with changing configurations, for CG benchmark ($D_{max} = 13$). Here, the *FAB* utilization is very low and remains so with varying configurations. The *Ready Deque* utilization stays close to 100% with varying configurations. Fig. 15 gives the variation of *FAB* space and *Ready Deque* space with changing configurations, for Heat benchmark ($D_{max} = 12$). Here, the *FAB* utilization is high (close to 100%) and remains so with varying configurations. The *Ready Deque* utilization also stays close to 100% with varying configurations. The space utilization in the *AMRejectMap* and the *WorkRejectMap* remains constant per place on average.This empirically demonstrates that our distributed scheduling algorithm has efficient space utilization and is in fact better ($O(m \cdot D_{max} \cdot S_{max})$ per place) compared to the theoretical bound established by Theorem 4.5.

5.1.1 Space Time Trade-Offs Analysis

We measured the detailed space-time trade-offs in our distributed scheduling algorithm on multi-core Power6 platform. This has 32 Power6 cores and total $128GB$ memory. Each core has $64KB$ instruction $L1$ cache and $64KB$ $L1$ data cache along with $4MB$ semi-private unified L2 cache. Two cores on a Power6 chip share an external $32MB$ $L3$ cache. The experiments were performed with 16 places and 4 processors per place, using 64 hardware threads on the 32 core Power6 machine.

Here, we decrease the available *FAB* space at each place from D_{max} stack frames, to 1 stack frame, and measure the corresponding increase in the total time, work stealing time and waiting time of an activity/thread. Both, the *Ready Deque* and *PrQ* (w/ *SAB*) space were kept constant at D_{max}. Due to bounded space per place and per processor, the activities have to wait before space becomes available for their execution. This *waiting time* is averaged over all waiting activities and then reported in the experiments.

Fig. 16 illustrates the increase in total time, work stealing time and waiting time for the MD benchmark (with parameters values: number of molecules = $32K$ and number of iterations = 100), with the variation of *FAB* space from 11 stack frames to 1 stack frame. The *Ready Deque* space and *PrQ* (with *SAB*) space are both kept constant at $D_{max} = 11$ stack frames. The total time increases from $347s$ to $406s$ (17% increase). This is due to increase in the work

stealing time from $29s$ to $43s$ and increase in the waiting time of the activities (threads), due to bounded *FAB* space and *PrQ* (and *SAB*) space, from $110s$ to $147s$.

For the CG benchmark (Fig. 17), the total time increase is only $4s$ with the reduction in *FAB* space from 6 to 1 stack frame (keeping both *Ready Deque* space and *PrQ* with *SAB* space constant at $D_{max} = 6$ stack frames). This is due to very limited increase in the work stealing time, $7s$, and negligible change in the waiting time of the activities.

Fig. 18 illustrates the increase in total time, work stealing time and waiting time for the Heat benchmark (with parameters values: matrix size = $1024K * 4K$, leafmaxcol = 8 and number of iterations = 100), with the variation of *FAB* space from 15 stack frames to 1 stack frame. The *Ready Deque* space and *PrQ* (with *SAB*) space are both kept constant at $D_{max} = 15$ stack frames. The total time increases from $587s$ to $676s$ (15% increase). This can be attributed to the increase in the work stealing time from $32s$ to $64s$ and increase in the waiting time of the activities (threads), due to bounded *FAB* space and *PrQ* (and *SAB*) space, from $109s$ to $213s$. Thus, our distributed scheduling algorithm demonstrates efficient space-time trade-offs.

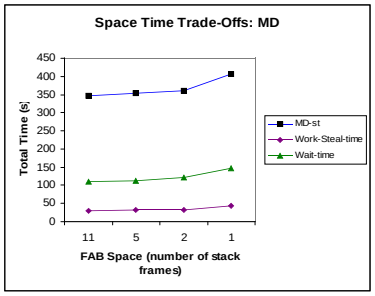

Figure 16. Space Time Trade-Offs: MD

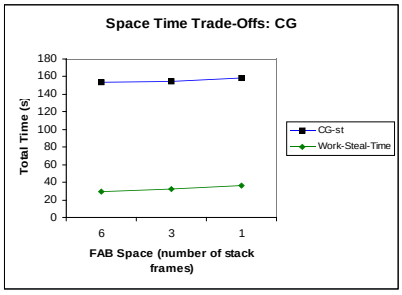

Figure 17. Space Time Trade-Offs: CG

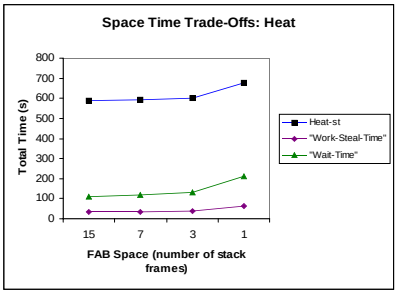

Figure 18. Space Time Trade-Offs: Heat

6. Conclusions & Future Work

We have addressed the challenging problem of affinity driven, online, deadlock-free distributed scheduling for parallel computations. Deadlock freedom proof and space bound analysis have been provided for our algorithm. On well known benchmarks our algorithm demonstrates around much better performance gains over typical Cilk style scheduling. Detailed experimental analysis shows the scalability of our algorithm along with efficient space-time tradeoffs. This is the first such work for affinity driven distributed scheduling of parallel computations in a multi-place setup. In future, we plan to look into markov-chain based modeling and multi-core cluster (such as Blue Gene/P) based implementation of our distributed scheduling algorithm.

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

[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] Bradford L. ChamberLain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291 – 312, August 2007.

[8] 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.*

[9] Exascale Study Group, Peter Kogge (Editor, Study Lead), and William Harrod (Program Manager). Exascale computing study: Technology challenges in achieving exascale systems. Technical report, Sep 2008.

[10] Jean-Nol Quintin and Frdric Wagner. Hierarchical work-stealing. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, pages 217–229. Springer-Verlag, 2010.

[11] Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec, and Bruno Raffi n. A work stealing scheduler for parallel loops on shared cache multicores. In *4th Workshop on Highly Parallel Processing on a Chip (HPPC)*, 2010.

[12] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In *The 21st International Symposium on Algorithms and Computation (ISAAC)*, 2010.

[13] Katherine Yelick and Dan Bonachea et.al. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.