# Affinity Driven Distributed Scheduling Algorithm for Parallel Computations

**Ankur Narang**[1], **Abhinav**[1], **Naga Praveen Kumar**[1], **and Rudrapatna K. Shyamasundar**[2]

[1] IBM Research - India, New Delhi
[2] Tata Institute of Fundamental Research, Mumbai

**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* [3] needs to follow affinity and deliver efficient space, time and message complexity. Simultaneous consideration of these factors makes affinity driven distributed scheduling particularly challenging. In this paper, we address this challenge by using a low time and message complexity mechanism for ensuring affinity and a randomized work-stealing mechanism within places for load balancing.
This paper presents an online algorithm for affinity driven distributed scheduling of *multi-place* [4] parallel computations. Theoretical analysis of the expected and probabilistic lower and upper bounds on time and message complexity of this algorithm has been provided. On well known benchmarks, our algorithm demonstrates $16\%$ to $30\%$ performance gain as compared to Cilk [6] on multi-core Intel Xeon 5570 architecture. Further, detailed experimental analysis shows the scalability of our algorithm along with efficient space utilization. To the best of our knowledge, this is the first time affinity driven distributed scheduling algorithm has been designed and theoretically analyzed in a *multi-place* setup for many core architectures.

## 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 [10]). 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 [9], Chapel [8] and Fortress [4] are based on partitioned global address space (PGAS [11]) paradigm. They have been designed and implemented as part of DARPA HPCS program [5] 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.

[3] place is a group of processors with shared memory
[4] multi-place refers to a group of places. For example, with each place as an SMP(Symmetric MultiProcessor), multi-place refers to cluster of SMPs
[5] www.highproductivity.org/

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.

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.

In this paper, we address the following affinity driven distributed scheduling problem. **Given: (a)** An input computation DAG (Fig. 1) that represents a parallel multi-threaded computation with fine to medium grained parallelism. 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 *strict* parallel computation DAG (synchronization dependency edge represents an activity waiting for the completion of a descendant activity, details in section 3); **(b)** A cluster of *n* SMPs (refer Fig. 2) as the target architecture on which to schedule the computation DAG. Each SMP [6] also referred as *place* has fixed number(*m*) of processors and memory. The cluster of SMPs is referred as the *multi-place* setup. **Determine:** An online schedule for the nodes of the computation DAG in a distributed fashion that ensures the following: **(a)** Exact mapping of nodes onto *places* as specified in the input DAG. **(b)** Low space, time and message complexity for execution.

In this paper, we present the design of a novel affinity driven, online, distributed scheduling algorithm with low time and message complexity. The algorithm assumes initial placement annotations on the given parallel computation with the consideration of load balance *across* the places. The algorithm controls the online expansion of the computation DAG. Our algorithm employs an efficient remote spawn mechanism across places for ensuring affinity. Randomized work stealing *within* a place helps in load balancing. Our main contributions are:

- We present a novel affinity driven, online, distributed scheduling algorithm. This algorithm is designed for strict multi-place parallel computations.
- Using theoretical analysis, we prove that the lower bound of 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

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

$k$, and $T_{\infty,n}$ denotes the execution time of the computation on $n$ places with infinite processors on each place. Expected and probabilistic lower and upper bounds for the message complexity have also been provided.

– On well known parallel benchmarks (Heat, Molecular Dynamics and Conjugate Gradient), we demonstrate performance gains of around $16\%$ to $30\%$ over Cilk on multi-core architectures. Detailed analysis shows the scalability of our algorithm as well as efficienct space utilization.

## 2 Related Work

Scheduling of dynamically created tasks for shared memory multi-processors has been a well studied problem. The work on Cilk [6] 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. [6] 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 $T_\infty$ 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. [5] 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) while ensuring affinity and also provide time and message complexity bounds.

[7] 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 that online greedy scheduling of multi-threaded computations may lead to physical deadlock in presence of bounded space and communication resources per place. However, the computation did not respect affinity always and 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 show empirically, that our algorithm has efficient space utilization as well.

## 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 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. 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: $(a)$ ***Fully-strict computation***: Dependencies are only between the nodes of a thread and the nodes of its immediate parent thread; $(b)$ ***Strict computation***: Dependencies are only between the nodes of a thread and the nodes of any of its ancestor threads; $(c)$ ***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 $\Gamma_r$, if there exists a dependence edge from any activity in the subtree to the root activity $\Gamma_r$, then there cannot exist any dependence edge from the activities in the subtree to the ancestors of $\Gamma_r$.

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. $T_\infty^k$ denotes the execution time for activities assigned to 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 time taken by a single processor for the activities assigned to place $k$.

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

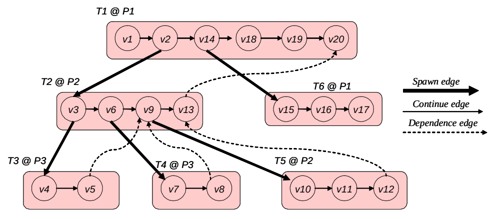


**Fig. 1.** Place-annotated Computation Dag

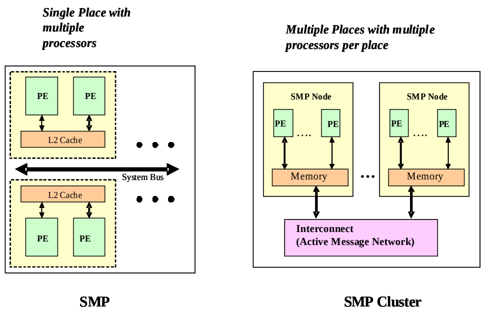

Single Place with multiple processors
PE
PE
L2 Cache
System Bus
L2 Cache
PE
PE
SMP
Multiple Places with multiple processors per place
SMP Node
PE
PE
Memory
SMP Node
PE
PE
Memory
Interconnect (Active Message Network)
SMP Cluster

**Fig. 2.** Multiple Places: Cluster of SMPs

## 4 Distributed Scheduling Algorithm

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. For space efficiency, before a place-annotated activity is pushed onto a place, the remote place buffer (*FAB*, see below) is checked for space utilization. If the space utilization of the remote buffer (*FAB*) is high then the push gets delayed for a limited amount of time. This helps in appropriate space-time tradeoff for the execution of the parallel computation. Within a place, the online unfolding of the computation DAG happens in a depth-first manner to enable efficient space and time execution. Sufficient space is assumed to exist at each place, so physical deadlocks due to lack of space cannot happen in this algorithm.

Each place maintains a *Fresh Activity Buffer* (*FAB*) which is managed by a dedicated processor (different from workers) at that place. An activity that has affinity for a remote place is pushed into the *FAB* at that place. Each worker at a place has a *Ready Deque* and a *Stall Buffer* (refer Fig. 3). The *Ready Deque* of a processor contains the activities of the parallel computation that are ready to execute. The *Stall Buffer* contains the activities that have been stalled due to dependency on another activities in the parallel computation. The *FAB* at each place as well as the *Ready Deque* at each worker use a concurrent deque implementation. 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 below.

At any step, an activity $A$ at the $r^{th}$ worker (at place $i$), $W_i^r$, may perform the 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. If the space utilization of *FAB*$_{(j)}$ is below a given thresh-

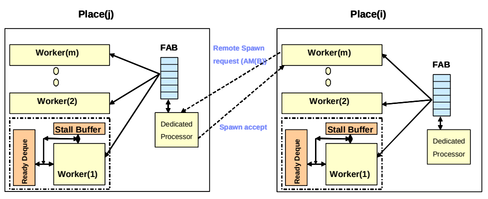


**Fig. 3.** Affinity Driven Distributed Scheduling Algorithm

old, then *AM(B)* is successfully inserted in the *FAB*$_{(j)}$ (at $P_j$) and $A$ continues execution. Else, this worker waits for a limited time, $\delta_t$, before retrying the activity $B$ spawn on place $P_j$ (Fig. 3).

(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 $B$): An activity, $A$, (after termination or otherwise) 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*.

### 4.1 Time Complexity Analysis

The time complexity of this affinity driven distributed scheduling algorithm in terms of number of *throws* during execution is presented below. Each *throw* represents an attempt by a worker(*thief*) to steal an activity from either another worker(*victim*) or *FAB* at the same place.

**Lemma 1.** *Consider a strict place-annotated computation DAG with work per place, $T_1^k$, being executed by the distributed scheduling algorithm presented in section 4. 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 $O(\max_k(T_1^k/m + Q_r^k/m))$ and the upper bound is $O(\sum_k(T_1^k/m + Q_r^k/m))$.*

*Proof Sketch:* (Token based counting argument) Consider three buckets at each place in which tokens are placed: *work bucket* where a token is placed when a worker at that place executes a node of the computation DAG; *ready-node-throw bucket* where a token is placed when a worker attempts to steal and there is at least one ready node at that place; *null-node-throw bucket* where a token is placed when a worker attempts to steal and there are no ready nodes at that place (models the wait time when there is no work at that place). The total finish time of a place can be computed by counting the tokens in these three buckets and by considering load balanced execution within a place (using randomized work stealing). The upper and lower bounds on the execution time arise from the structure of the computation DAG and the structure of the online schedule generated. The detailed proof is presented in [3].

Next, we compute the bound on the number of tokens in the ready-node-throw bucket using potential function based analysis [5]. Our unique contribution is in proving the lower and upper bounds of time complexity and message complexity for the **multi-place** affinity driven distributed scheduling algorithm presented in section 4 that involves **both** *intra-place work stealing* and *remote place affinity driven work pushing*.

Let there be a non-negative potential associated with each ready node in the computation dag. If the execution of node $u$ enables node $v$, then *edge(u,v)* is called the *enabling edge* and $u$ is called the *designated parent* of $v$. The subgraph of the computation DAG consisting of enabling edges forms a tree, called the *enabling tree*. During the execution of the affinity driven distributed scheduling algorithm 4, the weight of a node $u$ in the *enabling tree*, $w(u)$ is defined as $(T_{\infty,n} - depth(u))$. 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 $i$, $\phi_i$, is the sum of the potential of all the ready nodes 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 workers whose *Ready Deque* is empty at the beginning of step $i$, and let $D_i$ denote the set of all other workers with non-empty *Ready Deque*. Let, $F_i$ denote the set of all ready nodes present across the *FAB* at all places. The total potential can be partitioned into three parts as follows:

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

Actions such as assignment of a node from *Ready Deque* to the worker for execution, stealing nodes from the top of victim's *Ready Deque* and execution of a node, lead to decrease of potential. The idle workers at a place do work-stealing alternately between stealing from *Ready Deque* and stealing from the *FAB*. Thus, $2m$ throws in a round consist of $m$ throws to other workers's *Ready Deque* and $m$ throws to the *FAB*. For randomized work-stealing one can use the *balls and bins* game [3] to compute the expected and probabilistic bound on the number of throws. Using this, one can show that whenever $m$ or more throws occur for getting nodes from the top of the Ready Deque of other workers at the same place, the potential decreases by a constant fraction of $\phi_i(D_i)$ with a constant probability. The component of potential associated with the *FAB* at place $P_k$ , $\phi_i^k(F_i)$, can be shown to deterministically decrease for $m$ throws in a round. Furthermore, at each place the potential also drops by a constant factor of $\phi_i^k(E_i)$. The detailed analysis of decrease of potential for each component is given in [3]. Analyzing the rate of decrease of potential and using Lemma 1 leads to the following theorem.

**Theorem 1.** *Consider a strict place-annotated computation DAG with work per place $k$, denoted by $T_1^k$, being executed by the affinity driven multi-place distributed scheduling algorithm (section 4). Let the critical-path length for the computation be $T_{\infty,n}$. 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 lower bound for the execution time is $O(\max_k T_1^k/m + T_{\infty,n} + \log(1/\epsilon))$ with probability at least $1 - \epsilon$. Similar probabilistic upper bound exists.*

*Proof Sketch:* For the lower bound, we analyze the number of throws (to the ready-node-throw bucket) by breaking the execution into phases of $\theta(P = mn)$ throws ($O(m)$ throws per place). It can be shown that with constant probability, a phase causes the potential to drop by a constant factor. More precisely, between phases $i$ and $i + 1$, $Pr\{(\phi_i - \phi_{i+1}) \geq 1/4.\phi_i\} > 1/4$ (details in [3] ). Since the potential starts at $\phi_0 = 3^{2T_{\infty,n}-1}$ and ends at zero and takes integral values, the number of successful phases is at most $(2T_{\infty,n} - 1)\log_{4/3} 3 < 8T_{\infty,n}$. Thus, the expected number of throws per place gets bounded by $O(T_{\infty,n} \cdot m)$, and the number of throws is $O(T_{\infty,n} \cdot m) + \log(1/\epsilon))$ with probability at least $1 - \epsilon$ (using Chernoff Inequality). Using Lemma 1 we get the lower bound on the expected execution time as $O(\max_k T_1^k/m + T_{\infty,n})$. The detailed proof and probabilistic bounds are presented in [3] .

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 $O(T_\infty^k \cdot m)$. Further, the place that finishes the execution in the end, can end up with the number of tokens in the *null-node-throw bucket* equal to the tokens in the *work buckets* and the *read-node-throw buckets* of all other places. Hence, the finish time for this place, which is also the execution time of the full computation DAG is $O(\sum_k (T_1^k/m + T_\infty^k))$. The probabilistic upper bound can be similarly established using Chernoff Inequality.

The following theorem bounds the message complexity of the affinity driven work stealing algorithm 4.

**Theorem 2.** *Consider the execution of a strict place-annotated computation DAG with critical path-length $T_{\infty,n}$ by the Affinity Driven Distributed Scheduling Algorithm (section 4). Then, the total number of bytes communicated across places is $O(I \cdot (S_{max} + n_d))$ and the lower bound on number of bytes communicated within a place has the expectation $O(m \cdot T_{\infty,n} \cdot S_{max} \cdot n_d)$, where $n_d$ is the maximum number of dependence edges from the descendants to a parent and $I$ is the number of remote spawns from one place to a remote place. Moreover, for any $\epsilon > 0$, the probability is at least $(1-\epsilon)$ that the lower bound on the communication overhead per place is $O(m \cdot (T_{\infty,n} + \log(1/\epsilon)) \cdot n_d \cdot S_{max})$. Similarly message upper bounds exist.*

*Proof.* First consider inter-place messages. Let the number of affinity driven pushes to remote places be $O(I)$, each of maximum $O(S_{max})$ 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 $O(m.T_{\infty,n})$ with each steal attempt requiring $S_{max}$ 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' Ready Deque. Since this can happen $n_d$ times for each time the parent is stolen, the communication involved is at most $n_d.S_{max})$. So, the expected total intra-place communication across all places is $O(n.m.T_{\infty,n}.S_{max}.n_d)$. 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.

## 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*. Using well known benchmarks the performance of *ADS* was compared with *CWS* and also with original Cilk [8] scheduler (referred as *CORG* in this section). These benchmarks are the following. **Heat**: Jacobi over-relaxation that simulates heat propagation on a two dimensional grid for a number of steps [1]. For our scheduling algorithm (*ADS*), the 2D grid is partitioned uniformly across the available cores. [9]; **Molecular Dynamics** (MD): This is classical Molecular Dynamics simulation, using the Velocity Verlet time integration scheme. The simulation was carried on $16K$ particles for $100$ iterations; **Conjugate Gradient** (NPB [10] 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 between *ADS* and *CORG* was done on Intel multi-core platform. This 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. Fig. 4 compares the performance for the Heat benchmark (matrix: $32K * 4K$, number of iterations = $100$, leafmaxcol = $32$). Both *ADS* and *CORG* demonstrate strong scalability. Initially, *ADS* is around $1.9\times$ better than *CORG*, but later this gap stabilizes at around $1.20\times$.

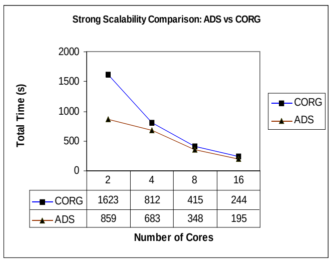
### 5.1 Detailed Performance Analysis

In this section, we analyze the performance gains obtained by our *ADS* algorithm vs. the Cilk style scheduling (*CWS*) algorithm and also investigate the behavior of our algorithm on Power6 multi-core architecture.

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

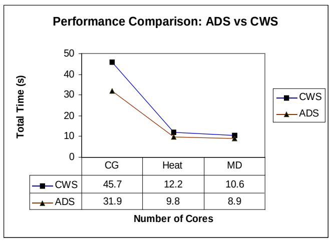
[9] The $D_{max}$ for this benchmark is $\log(numCols/leafmaxcol)$ where *numCols* represents the number of columns in the input two-dimensional grid and *leafmaxcol* represents the number of columns to be processed by a single thread

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



| | 2 | 4 | 8 | 16 |
|---|---|---|---|---|
| CORG | 1623 | 812 | 415 | 244 |
| ADS | 859 | 683 | 348 | 195 |

**Fig. 4.** CORG vs ADS



| | CG | Heat | MD |
|---|---|---|---|
| CWS | 45.7 | 12.2 | 10.6 |
| ADS | 31.9 | 9.8 | 8.9 |

**Fig. 5.** ADS vs CWS

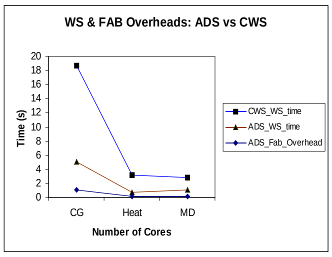


**Fig. 6.** ADS vs CWS

Fig. 5 demonstrates the gain in performance of *ADS* vs *CWS* with 16 cores. For CG, Class B matrix is chosen with parameters: *NA* = $75K$, *Non-Zero* = $13M$, *Outer iterations* = $75$, *SHIFT* = $60$. For Heat, the parameters values chosen are: matrix size = $32 * 4K$, number of iterations = $100$ and *leafmaxcol* = $32$. While CG has maximum gain of $30\%$, MD shows gain of $16\%$. Fig. 6 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* ($5s$) is $3.74\times$ better than *CWS* ($18.7s$). For Heat and MD, *ADS* work steal time is $4.1\times$ and $2.8\times$ better respectively, as compared to *CWS*. *ADS* has *FAB* overheads but this time is very small, around $13\%$ to $22\%$ 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*.

We measured the detailed characteristics of our scheduling algorithm on multi-core Power6 platform. This has 16 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. Fig. 7 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 $= 16$, the configurations, in the format *(number of places * number of processors per place)*, chosen are: $(a)$ $(16 * 1)$, $(b)$ $(8 * 2)$, $(c)$ $(4 * 4)$, and $(d)$ $(2 * 8)$. As the number of places increase from 2 to 8, the work steal time increases from $3.5s$ to $80s$ as the average number of work steal attempts increases from $140K$ to $4M$. 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 $0.3s$ for 2 places, to $110s$ 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 ($300M$ average) is very large, while the successful FAB attempts are very low (1400 average). With increasing number of places from 2 to 16, the total time increases from $189s$ to $425s$, due to increase in work stealing and/or FAB steal overheads.

Fig. 8 plots the work stealing time and *FAB* stealing time variation with changing multi-place configurations for the CG benchmark (using Class C matrix with parameter values: *NA* = $150K$, *Non-Zero* = $13M$, *Outer Iterations* = $75$ and *SHIFT* = $60$). In this case, the work steal time increases from $12.1s$ (for $(2*8)$) to $13.1$ (for $(8*2)$) and then falls to $0$ for $(16*1)$ configuration. The *FAB* time initially increases slowly from $3.6s$ to $4.1s$ but then jumps to $81s$ for $(16*1)$ configuration. This behavior can be explained as in the case of MD benchmark (above).

Fig. 9 plots the work stealing time and *FAB* stealing time variation with changing multi-place configurations for the CG benchmark (using parameter values: *matrix size* = $64K * 8K$, *Iterations* = $100$ and *leafmaxcol* = 32). The variation of work stealing time, FAB stealing time and total time follow the pattern as in the case of MD.

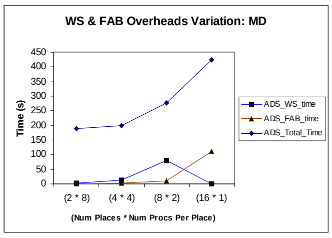


**Fig. 7.** Overheads - MD

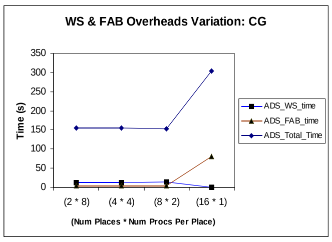


**Fig. 8.** Overheads - CG

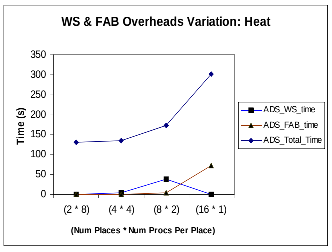


**Fig. 9.** Overheads - HEAT

Fig. 10 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 $11$ stack frames, which leads to $81\%$ maximum *FAB* utilization and roughly $109\%$ *Ready Deque* utilization. Fig. 12 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. 11 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. This empirically demonstrates that our distributed scheduling algorithm has efficient space utilization as well.

## 6 Conclusions & Future Work

We have addressed the challenging problem of affinity driven online distributed scheduling of parallel computations. We have provided theoretical analysis of the time and message complexity bounds of our algorithm. On well known benchmarks our algorithm demonstrates around $16\%$ to $30\%$ performance gain over typical Cilk style scheduling.

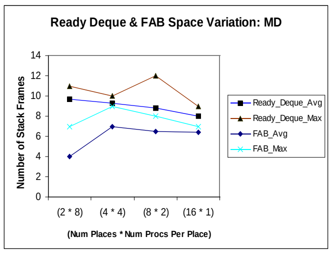


Fig. 10. Space Util - MD

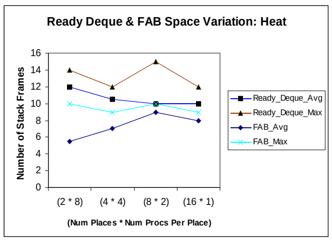


Fig. 11. Space Util - HEAT

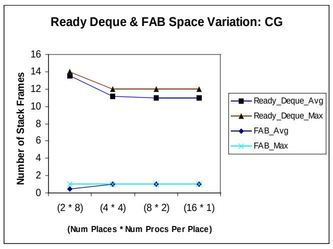


Fig. 12. Space Util - CG

Detailed experimental analysis shows the scalability of our algorithm along with efficient space utilization. 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 space-time tradeoffs and markov-chain based modeling of the distributed scheduling algorithm.

## References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: SPAA. pp. 1 – 12. New York, NY, USA (December 2000)
2. Agarwal, S., R.Barik, Bonachea, D., Sarkar, V., Shyamasundar, R.K., Yellick, K.: Deadlock-free scheduling of x10 computations with bounded resources. In: SPAA. pp. 229 – 240. San Diego, CA, USA (December 2007)
3. Agarwal, S., Narang, A., Shyamasundar, R.K.: Affinity driven distributed scheduling algorithms for parallel computations. Tech. Rep. RI09010, IBM India Research Labs, New Delhi (July 2009)
4. Allan, E., Chase, D., Luchangco, V., Maessen, J.W., Ryu, S., Jr., G.L.S., Tobin-Hochstadt, S.: The Fortress language specification version 0.618. Tech. rep., Sun Microsystems (apr 2005)
5. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: SPAA. pp. 119 – 129. Puerto Vallarta, Mexico (1998)
6. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)
7. Blumofe, R.D., Lisiecki, P.A.: Adaptive and reliable parallel computing on networks of workstations. In: USENIX Annual Technical Conference. Anaheim, California (1997)
8. ChamberLain, B.L., Callahan, D., Zima, H.P.: Parallel Programmability and the Chapel Language. International Journal of High Performance Computing Applications 21(3), 291 – 312 (August 2007)
9. Charles, P., Donawa, C., Ebcioglu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: OOPSLA 2005 Onward! Track (2005)
10. Group, E.S., (Editor, P.K., Lead), S., Manager), W.H.P.: Exascale computing study: Technology challenges in achieving exascale systems. Tech. rep. (Sep 2008)
11. Yelick, K., et.al., D.B.: Productivity and performance using partitioned global address space languages. In: PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation. pp. 24–32. ACM, New York, NY, USA (2007)