Performance Driven Multi-Objective Distributed Scheduling for Parallel Computations

Ankur Narang Abhinav Srivastava
Naga Praveen Kumar Katta
IBM Research - India, New Delhi
annarang@in.ibm.com, {abhinavsriva, praveen2005.iitk}@gmail.com

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

Abstract

With the advent of many-core architectures and strong need for Petascale (and Exascale) performance in scientific domains and industry analytics, efficient scheduling of parallel computations for higher productivity and performance has become very important. Further, movement of massive amounts (Terabytes to Petabytes) of data is very expensive, which necessitates affinity driven computations. Therefore, distributed scheduling of parallel computations on multiple *places* [1] needs to optimize multiple performance objectives: follow affinity maximally and ensure efficient space, time and message complexity. Simultaneous consideration of these objectives makes distributed scheduling a particularly challenging problem. In addition, parallel computations have data dependent execution patterns which requires *online* scheduling to effectively optimize the computation orchestration as it unfolds.

This paper presents an online algorithm for affinity driven distributed scheduling of *multi-place* [2] parallel computations. To optimize multiple performance objectives simultaneously, our algorithm uses a low time and message complexity mechanism for ensuring affinity and a *randomized work-stealing* mechanism within places for load balancing. Theoretical analysis of the expected and probabilistic lower and upper bounds on time and message complexity of this algorithm has been provided. On multi-core clusters such as Blue Gene/P (MPP architecture) and Intel multi-core cluster, we demonstrate performance close to the custom MPI+Pthreads code. Further, strong, weak and data (increasing input data size) scalability have been demonstrated on multi-core clusters. Using well known benchmarks, we demonstrate 16% to 30% performance gain as compared to Cilk [6] on multi-core Intel Xeon 5570 (NUMA) architecture. Detailed experimental analysis illustrates efficient space (main memory) utilization as well. To the best of our knowledge, this is the first time multi-objective affinity driven distributed scheduling algorithm has been designed, theoretically analyzed and experimentally evaluated in a *multi-place* setup for multi-core cluster 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 [3] are based on Partitioned Global Address Space (*PGAS* [13]) paradigm. They have been designed and implemented as part of DARPA HPCS program [3] for higher productivity and performance on many-core massively parallel platforms. These languages have built-in support for initial placement of threads and data structures in the parallel program. Therefore, locality (affinity) 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. Centralized scheduling algorithms suffer from unnecessary overheads and non-scalable performance. Distributed scheduling avoids these pitfalls of centralized scheduling approaches. 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. This is especially true for data-dependent computations where static analysis based optimizations in the compiler cannot help much for performance driven scheduling.

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

[2] 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

[3] www.highproductivity.org/

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. As the computation graph unfolds in time, the online scheduler has to make decisions dynamically on where (which place and processor) and when (order) to schedule the computations.

Moreover, in order to mitigate the communication overheads in scheduling and lower data access and synchronization costs in the given parallel computation, it is essential to follow *affinity* inherent in the computation. For large scale computations involving processing petabytes to exabytes of data, affinity driven scheduling is necessary for scalability and superior performance. Further, the critical path of the scheduled computation is dependent on load balancing across the cores as well as on time and message complexity. Along with this, the space (main memory) consumed by the scheduler should be small enough to allow for large computations. Simultaneous consideration of these objectives: affinity, time, space 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 (instruction) 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:

- Spawn of a new thread.
- Sequential flow of execution.
- Synchronization dependency between two nodes.

The DAG is a *strict* parallel computation DAG (synchronization dependency edge represents a thread waiting for the completion of a descendant thread, 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 [4] 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.

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

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. [11] presents an affinity distributed scheduling algorithm with high level theoretical results and experimental analysis limited to Intel shared memory NUMA architecture. This work is an extension of [11] with detailed theoretical analysis and also extensive experimental analysis on multi-core clusters including Intel multi-core cluster as well as Blue Gene/P with upto 256 places. Our main contributions are:

- We present a novel multi-objective affinity driven, online, distributed scheduling algorithm. This algorithm is designed for strict multi-place parallel computations.
- Using detailed 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 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 multi-core cluster architectures such as Blue Gene/P (MPP architecture) as well as Intel multi-core clusters, we demonstrate that the performance of our distributed scheduling algorithm is close to custom (hand-written) MPI+Pthreads code. Further, strong, weak and data (increase in input size) scalability on multi-core clusters has been demonstrated. Using well known parallel benchmarks (Heat, Molecular Dynamics and Conjugate Gradient), we demonstrate performance gains of around 16% to 30% over Cilk on multi-core Intel (NUMA) architecture. Detailed analysis illustrates efficient space (main memory) utilization as well.

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_∞ is the execution time on infinite processors. We consider locality oriented scheduling in distributed environments and hence are more general than Cilk, and we also provide detailed time and message complexity analysis.

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) while ensuring affinity and also provide detailed analysis of 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 (main memory) utilization as well.

KAAPI [5] ("Kernel for Adaptive, Asynchronous Parallel and Interactive programming") is a C++ library that allows execution of fine/medium grain multi-threaded computations with dynamic data flow synchronizations. However, its performance for large number of places is not clearly specified. We demonstrate multi-place performance on Blue Gene/P architecture as well as Intel Cluster and show that it is close to custom MPI+Pthreads code. [6] is a C++ based parallel programming system (Charm++) that implements a message-driven migratable objects programming model, supported by an adaptive runtime system. It is based on a message-driven migratable objects programming model, and consists of a C++-based parallel notation, an adaptive runtime system (RTS) that automates resource management, a collection of debugging and performance analysis tools, and an associated family of higher level languages. It has been used to program several highly scalable parallel applications. This programming model is different from the Cilk multi-threaded programming model which we also leverage.

[5] proves stability of the basic Cilk-style work stealing mechanism using markov chain modeling and multi-step analysis along with Lyapunov function based arguments. [12] presents a general methodology for computing the expected makespan based on the analysis of an adequate potential function which represents the load unbalance between the local lists. A bound on the deviation from the mean is also derived. Then, this technique is applied to show that the expected makespan for scheduling W unit independent tasks on m processors is equal to W/m with an additional term in $3.65\log(W)$. While both the above efforts are for a single place, we consider multiple places and provide potential function based analysis to derive expected and probabilistic lower and upper bounds on time and message complexity for our affinity driven algorithm.

[11] presents an affinity distributed scheduling algorithm with high level theoretical results and experimental analysis limited to Intel shared memory NUMA architecture. This work is an extension of [11] with detailed theoretical analysis and also extensive experimental analysis on multi-core clusters including Intel multi-core cluster as well as Blue Gene/P with upto 256 places.

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.

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 (as in a processor instruction set architecture) like *and*, *or*, *not*, *add* and so forth. There are edges between the nodes (basic instructions such as and/or/add etc) in the computation

[5] https://gforge.inria.fr/projects/kaapi/

[6] http://charm.cs.uiuc.edu/

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

DAG (Fig. 1) that either represent: (a) creation of new activities (*spawn* edge), (b) sequential execution flow between the nodes within a thread/activity (*continue* edge) and (c) synchronization dependencies (*dependence* edge) between the nodes. In the paper, we refer to the parallel computation over nodes (basic instructions such as and/add/or) to be scheduled as the *computation DAG*. At a higher level, the parallel computation can also be viewed as a computation tree of *threads*. Each *thread* (as in multi-threaded programs) is a sequential flow of execution of instructions and consists of a set of nodes (basic operations/instructions). Each thread 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 threads 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). Here, the dependencies arise due to a thread waiting for the completion of its descendants. Every dependency edge, therefore, goes from the last instruction of a thread to one of its ancestor threads with the following restriction: In a subtree rooted at a thread called Γ_r, if there exists a dependence edge from any thread in the subtree to the root thread Γ_r, then there cannot exist any dependence edge from the threads in the subtree to the ancestors of Γ_r. Intuitively speaking, it does not allow the synchronization edges to cross each other.

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 cores (processors) at place P_i. D_{max} denotes the maximum depth of the computation tree in terms of number of threads. $T_{\infty,n}$ 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, $T_{\infty,n} \leq \sum_{1 \leq k \leq n} T_\infty^k$. T_1^k denotes the time taken by a single processor for the activities assigned to place k.

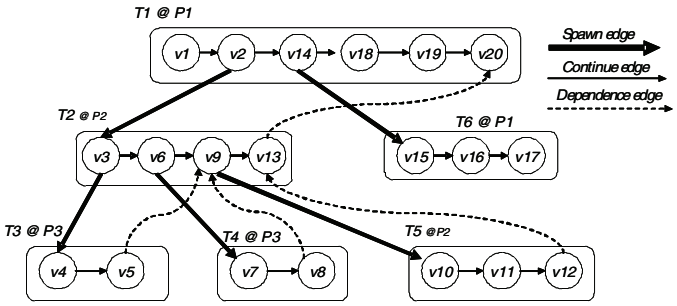

Figure 1. Place-annotated Computation Dag

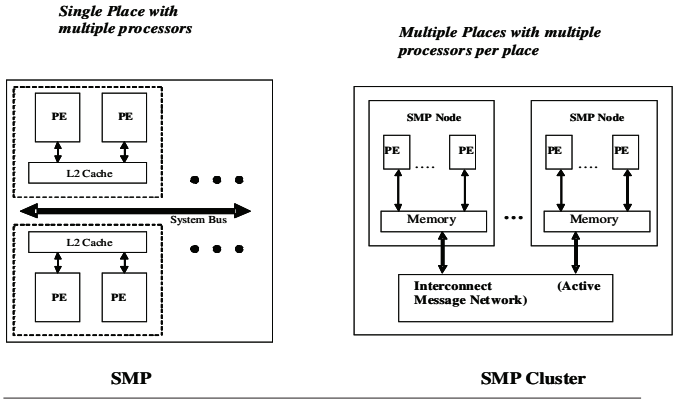

Figure 2. Multiple Places: Cluster of SMPs

4. Distributed Scheduling Algorithm

Consider a *strict* place-annotated computation DAG. The distributed scheduling algorithm described below schedules threads 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 threads are pushed onto their respective remote places. For space (main memory) efficiency, before a place-annotated thread is pushed onto a place, the remote place buffer (*FAB*, see below) is checked for space utilization. Here, space utilization refers to occupancy of the FAB buffer. If the space utilization (occupancy) of the remote buffer (*FAB*) is high i.e. greater than a threshold say 0.5, then the push gets delayed for a limited amount of time. This helps in appropriate space-time trade-off 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. A thread that is annotated with 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 threads of the parallel computation that are ready to execute. The *Stall Buffer* contains the threads that have been stalled due to dependency on another threads 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 processor at a place will attempt to randomly steal work from other processors at the same place (*randomized work stealing*). Note that a thread which is pushed onto a place can move between

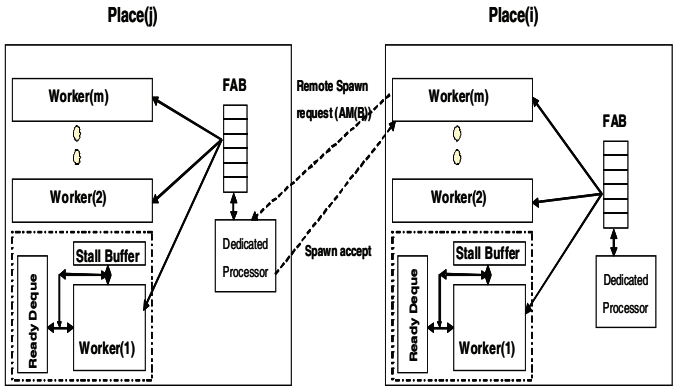

Figure 3. Affinity Driven Distributed Scheduling Algorithm

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, a thread A at the r^{th} worker (at place i), W_i^r, may perform the following actions:

1. **Spawn**:
   (a) A spawns thread 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 threshold, 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, δ_t, before retrying the thread 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 a thread 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 thread from the *FAB* at that place. If there is no thread in the *FAB* then another victim worker is chosen from the same place.
3. **Stalls** (A stalls): A thread 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): A thread, A, (after termination or otherwise) may enable a stalled thread 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 4.1. *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 At any place, k, we collect tokens in three buckets: *work bucket*, *ready-node-throw bucket* and *null-node-throw bucket*. In the work bucket the tokens get collected when the processors at the place k execute the ready nodes. Thus, the total number of tokens collected in the work bucket is T_1^k. When, the place has some ready nodes and a processor at that place throws or attempts to steal ready nodes from the PrQ or another processor's deque then the tokens are added to the read-node-throw bucket. If there are no ready nodes at the place then the throws by processors at that place are accounted for by placing tokens in the null-node-throw bucket. The tokens collected in these three buckets account for all work done by the processors at the place till the finish time for the computation at that place. Thus, the finish time at the place k, is $O(T_1^k/m + Q_r^k/m + Q_e^k/m)$. The finish time of the complete computation DAG is the maximum finish time over all places. So, the execution time for the computation is $\max_k O(T_1^k/m+Q_r^k/m+Q_e^k/m)$. We consider two extreme scenarios for Q_e^k that define the lower and upper bounds. For the lower bound, at any step of the execution, every place has some ready node, so there are no tokens placed in the null-node-throw bucket at any place. Hence, the execution time per place is $O(T_1^k/m + Q_r^k/m)$. The execution time for the full computation becomes $O(\max_k(T_1^k/m + Q_r^k/m))$. For the upper bound, there exists a place, say (w.l.o.g.) s, where the number of tokens in the null-node-throw buckets, Q_e^s, is equal to the sum of the total number of tokens in the work buckets of all other places and the tokens in the read-node-throw bucket over all other places. Thus, the finish time for this place, T_f^s, which is also the execution time for the computation is given by:

$$T_f^s = O(\sum_{1 \leq k \leq n} (T_1^k/m + Q_r^k/m)) \qquad (4.1)$$

□

Next, we compute the bound on the number of tokens in the ready-node-throw bucket using potential function based analysis [4]. Our unique contribution is in proving the lower and upper bounds of time complexity and message complex-

ity 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 (section 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:

$$\phi_i(u) = 3^{2w(u)-1}, \text{if u is assigned;} \tag{4.2a}$$

$$= 3^{2w(u)}, \text{otherwise} \tag{4.2b}$$

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

$$\phi_i = \phi_i(E_i) + \phi_i(D_i) + \phi_i(F_i) \tag{4.3}$$

where,

$$\phi_i(E_i) = \sum_{q \epsilon E_i} \phi_i(q) = \sum_{1 \le k \le n} \phi_i^k(E_i); \tag{4.4a}$$

$$\phi_i(D_i) = \sum_{q \epsilon D_i} \phi_i(q) = \sum_{1 \le k \le n} \phi_i^k(D_i); \tag{4.4b}$$

$$\phi_i(F_i) = \sum_{q \epsilon F_i} \phi_i(q) = \sum_{1 \le k \le n} \phi_i^k(F_i); \tag{4.4c}$$

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

$$\phi_i^k = \phi_i^k(E_i) + \phi_i^k(D_i) + \phi_i^k(F_i) \tag{4.5}$$

Actions such as assignment of a node from deque to the processor for execution, stealing nodes from the top of victim's deque and execution of a node, lead to decrease of potential (refer Lemma 4.3). The idle processors at a place do work-stealing alternately between stealing from deque and stealing from the FAB. Thus, $2P$ throws in a round consist of P throws to other processor's deque and P throws to the FAB. We first analyze the case when randomized work-stealing takes place from another processor's deque using *balls and bins* game to compute the expected and probabilistic bound on the number of throws. For uniform and random throws in the balls and bins game it can be shown that one is able to get a constant fraction of the reward with constant probability (refer Lemma 4.4). The lemma below states that whenever P or more throws occur for getting nodes from the top of the deques of other processors at the same place, the potential decreases by a constant fraction of $\phi_i(D_i)$ with a constant probability. For our distributed scheduling algorithm (section 4), $P = m$ (only intra-place work stealing). The following lemmas: Lemma 4.2, Lemma 4.3 and Lemma 4.4 are all taken from [4].

LEMMA 4.2. *Consider any round i and any later round j, such that at least P throws have taken place between round i (inclusive) and round j (exclusive), then, $Pr\{(\phi_i - \phi_j) \ge 1/4.\phi_i(D_i)\} > 1/4$*

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

$$Pr\{(\phi_i - \phi_j) \ge (1/4.\phi_i(D_i) + \phi_{i \to j}^k(out))\} > 1/4 \tag{4.6}$$

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

$$\phi_j^k(F_j) - \phi_i^k(F_i) = \phi_{i \to j}^k(in), \tag{4.7}$$

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

$$\{\phi_i^k(E_i) - \phi_j^k(E_j)\} \ge 5/9.\phi_i^k(E_i). \tag{4.8}$$

LEMMA 4.3. *The potential function satisfies the following properties*

1. *When node u is assigned to a process at step i, then the potential decreases by at least $2/3\phi_i(u)$.*
2. *When node u is executes at step i, then the potential decreases by at least $5/9\phi_i(u)$ at step i.*
3. *For any process, q in D_i, the topmost node u in the deque for q maintains the property that: $\phi_i(u) \ge 3/4\phi_i(q)$*

4. If the topmost node u of a processor q is stolen by processor p at step i, then the potential at the end of step i decreases by at least $1/2\phi_i(q)$ due to assignment or execution of u.

LEMMA 4.4. Balls and Bins Game: *Suppose that at least P balls are thrown independently and uniformly at random into P bins, where for i = 1,2...P, bin i has weight W_i. The total weight $W = \sum_{1 \leq i \leq P} W_i$. For each bin i, define a random variable, X_i as,*

$$X_i = W_i, \textit{if some ball lands in bin } i \tag{4.9a}$$

$$= 0, \textit{otherwise} \tag{4.9b}$$

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

THEOREM 4.5. *Consider any place-annotated multi-threaded computation with total work T_1 and work per place denoted by T_1^k, being executed by the affinity driven multi-place distributed scheduling algorithm 4. Let the critical-path length for the computation be T_∞. The lower bound on the expected execution time is $O(\max_k T_1^k/m + T_{\infty,n})$ and the upper bound is $O(\sum_k (T_1^k/m + T_\infty^k))$. Moreover, for any $\epsilon > 0$, the execution time is $O(\max_k T_1^k/m + T_{\infty,n} + \log(1/\epsilon))$ with probability at least $1 - \epsilon$.*

Proof Lemma 4.1 provides the lower bound on the execution time in terms of number of throws. We shall prove that the expected total number of throws per place is $O(T_{\infty,n} \cdot m)$, and that the total number of throws per place is $O(T_{\infty,n} \cdot m + \log(1/\epsilon))$ with probability at least $1 - \epsilon$.

We analyze the number of ready-node-throws by breaking the execution into phases of $\theta(P = mn)$ throws ($O(m)$ throws per place). We show that with constant probability, a phase causes the potential to drop by a constant factor, and since we know that the potential starts at $\phi_0 = 3^{2T_{\infty,n}-1}$ and ends at zero, we can use this fact to analyze the number of phases. The first phase begins at step $t_1 = 1$ and ends at the first step, t_1', such that at least P throws occur during the interval of steps $[t_1, t_1']$. The second phase begins at step $t_2 = t_1' + 1$, and so on.

Combining equations (4.6), (4.7) and (4.8) over all places, the components of the potential at the places corresponding to $\phi_{i-j}^k(out)$ and $\phi_{i-j}^k(in)$ cancel out. Using this and Lemma 4.2, we get that: $Pr\{(\phi_i - \phi_j) \geq 1/4.\phi_i\} > 1/4$.

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

Now, using Lemma 4.1, we get that the lower bound on the expected execution time for the affinity driven multi-place distributed scheduling algorithm is $O(\max_k T_1^k/m + T_{\infty,n})$.

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

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

THEOREM 4.6. *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. We also implemented custom (referred as *Custom*) optimized MPI + Pthreads code for Heat and MD benchmarks to be run on multi-core clusters. 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). The benchmarks used 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 places (SMPs). [9]. The Jacobi over-relaxation algorithm runs for N_t iterations. For all implementations (*Custom*, *ADS* and *CWS*), the computation tree consists of threads that seek to divide the underlying matrix further as compared to the parent thread until the number of cols reaches *leafmaxcol*.
- **Molecular Dynamics** (MD): This is classical Molecular Dynamics simulation, using the Velocity Verlet time integration scheme. The simulation was carried on varying number of molecules from $4K$ to $32K$ molecules for 100 iterations. For data partitioning, the whole region where all molecules are located, is divided into as many parts as the number of places (SMPs). Each place performs force calculations and position updates for the molecules in its region. In each iteration, the force on each molecule from other molecules in the same region are performed. As the iterations proceed, molecules can move around to neighbor regions.
- **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.

We compared *Strong* scalability, *Weak* scalability and *Data* scalability between *ADS* and *Custom*. *Strong* scalability refers to the reduction in time with increase in the number of cores/processors used while keeping the input data size constant (input data size refers to *matrix size* in case of Heat and to *number of molecules* in case of MD). *Weak* scalability refers to variation in time when both the input data size as well as the number of cores/processors are increased in same proportions. *Data* scalability refers to increase of time with increase in the input data size while keeping the number of cores/processors the same.

5.1 Performance Comparison on Multi-core Clusters

The performance comparison between *Custom* and *ADS* was done on Blue Gene/P [11] (MPP architecture) as well as on Intel multi-core cluster. Each node (place) in Blue Gene/P is a quad-core chip with frequency of 850 MHz having 2 GB of DRAM and 32 KB of L1 cache per core. Nodes (places) are interconnected by a 3D torus interconnect (3.4 Gbps per link in each of the six directions) apart from separate collective and global barrier networks. The Intel multi-core cluster has Intel 8-core Xeon 5504 per place (SMP) and multiple places (SMPs) are connected by 1 Gbps Ethernet.

5.1.1 Performance Comparison for Heat

Fig. 4 demonstrates the strong scalability of *ADS* on large number of places (with 3 threads (cores) per place, including the communication thread) on Blue Gene/P. *ADS* achieves $126\times$ speedup with $128\times$ increase in number of places, which results in parallel efficiency of around 98%, while *Custom* code achieves $118\times$ speedup resulting in parallel efficiency of 92%. Further, *ADS* performance is within 6.7% of the performance of *Custom* and the difference in performance reduces to zero with increasing number of places from 2 to 256 places.

Fig. 5 demonstrates the weak scalability of *ADS* on large number of places (with 3 threads (cores) per place, including the communication thread) on Blue Gene/P. *ADS* incurs only $1.1\times$ increase of time with $8\times$ increase in number of places and input data size (matrix size increases from (64K X 4K) to (512K X 4K)). Further, *ADS* performance is within 4% of the performance of *Custom*.

Fig. 6 demonstrates the data scalability of *ADS* with increase in N_x from 64K to 512K on 256 places of BG/P, with $N_y = 4K$ and $N_t = 100$. *ADS* incurs $7.8\times$ increase in time with $8\times$ increase in data, which illustrates linear data scalability. Further, *ADS* performance is within 3% of the performance of *Custom* even for large matrix sizes (512K X 4K).

Fig. 7 demonstrates strong scalability of *ADS* for increasing number of threads per place (with total four places) on the Intel multi-core cluster. *ADS* achieves around $6.4\times$ speedup with $6\times$ increase in number of threads (cores) per SMP. This super-linear speedup is due to decrease in overheads per place with increasing number of threads per place. *Custom* also achieves super-linear speedup of $7.4\times$. Further, *ADS* performance is within 18.5% of the performance of *Custom*.

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

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

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

[11] http://www.research.ibm.com/bluegene

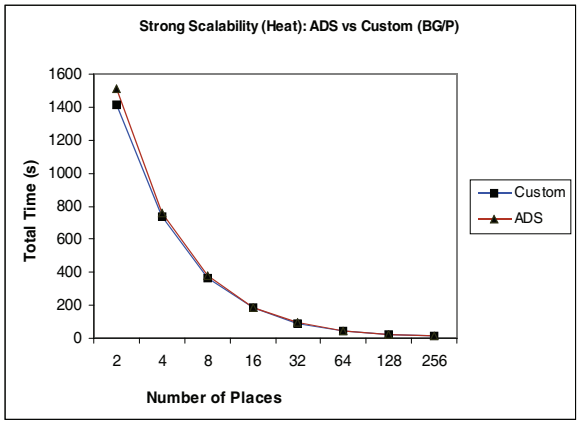

Figure 4. Heat: Strong Scalability (BG/P)

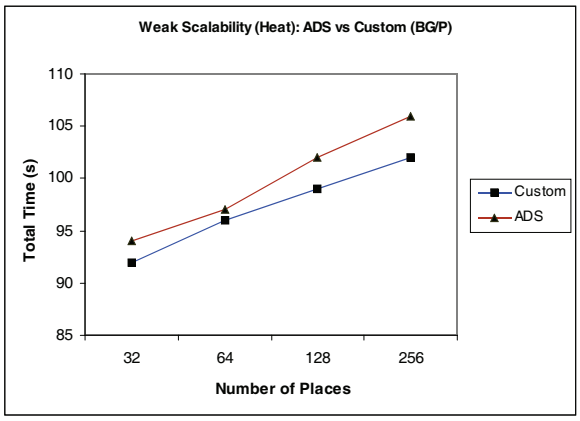

Figure 5. Heat: Weak Scalability (BG/P)

Fig. 8 demonstrates weak scalability of *ADS* for increasing number of threads per place (with two places) on the Intel multi-core cluster along with proportionate increase of input matrix size (from (12K X 4K) to (72K X 4K)). *ADS* incurs around $5\times$ *decrease* of time with $6\times$ increase in number of threads per place and data and *Custom* also incurs a similar ($5.3\times$) decrease in time. This is due to reduction in computation tree processing overheads and increase in cache performance with the increase in the number of threads per place. Further, *ADS* performance is within 9.2% of the performance of *Custom*.

Fig. 9 demonstrates the data scalability of *ADS* on large input data (matrix size) on the Intel multi-core cluster, for 6 threads per place and 4 places (SMPs) along with $N_y = 4$K and $N_t = 100$. *ADS* incurs around $8.6\times$ increase in time with $8\times$ increase in data (N_x increases from 32K to 256K), while *Custom* incurs $9.3\times$ increase in time. However, there is a sudden increase in time ($3\times$) as N_x increases from 256K to 512K. This is due to fall in cache performance with $N_x = 512$K. Further, *ADS* performance is within 4% of the performance of *Custom* for large matrix sizes (512K X 4K).

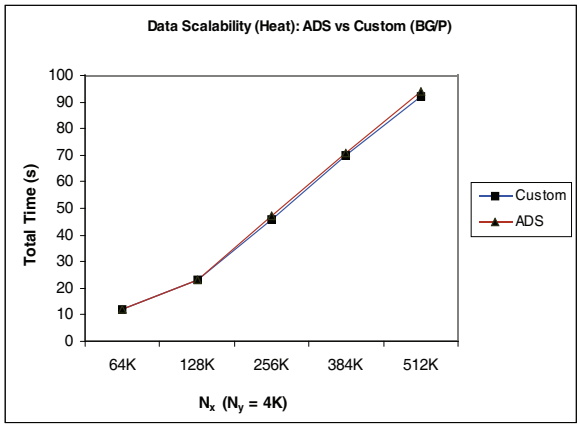

Figure 6. Heat: Data Scalability (BG/P)

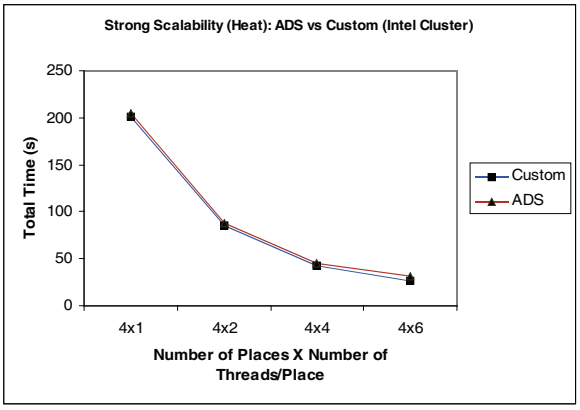

Figure 7. Heat: Strong Scalability (Intel Cluster)

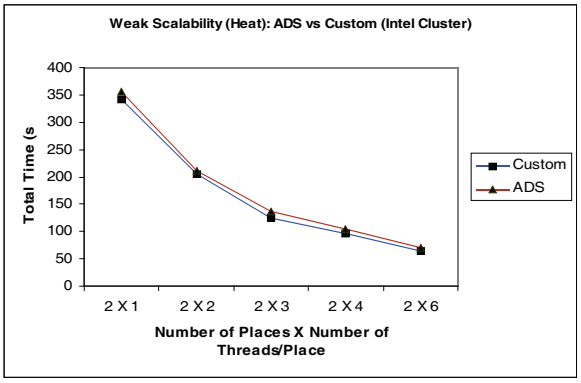

Figure 8. Heat: Weak Scalability (Intel Cluster)

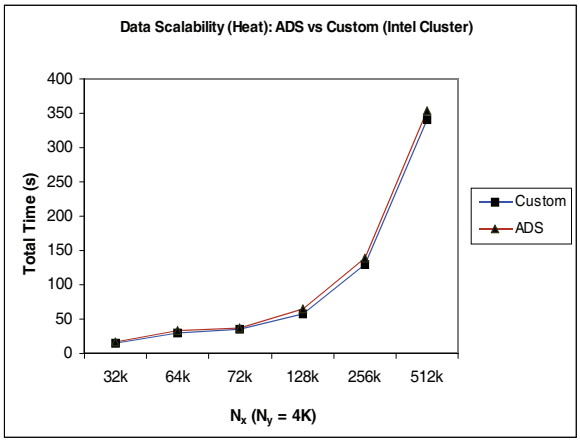

Figure 9. Heat: Data Scalability (Intel Cluster)

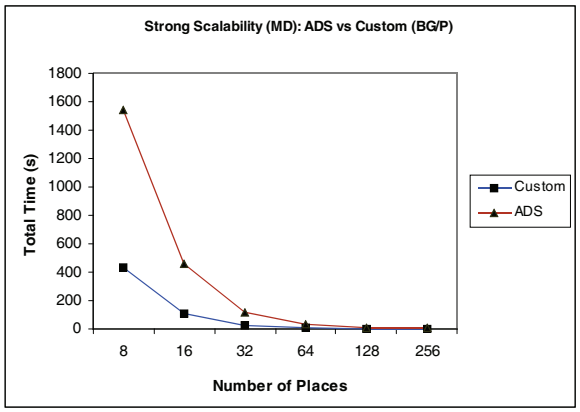

Figure 10. MD: Strong Scalability (BG/P)

5.1.2 Performance Comparison for MD

Fig. 10 demonstrates the strong scalability of *ADS* for increasing number of places, from 8 to 256 places. *ADS* achieves a *super-linear* speedup of around $219\times$ with $32\times$ increase in number of nodes; while *Custom* achieves $874\times$ speedup for the same increase in nodes. The reason for the super-linear speedup for both *ADS* and *Custom* is that with the smaller number of nodes (places), all molecules are on a single place which contains a single region for calculating forces between the molecules and hence more (quadratically larger) computations are performed. For larger number of nodes (places), such as 256, the molecules have spread around to 256 regions due to which the number of force calculations have gone down quadratically.

Fig. 11 demonstrates the weak scalability of *ADS* with increasing number of nodes and N_x on BG/P. *ADS* has around $3.3\times$ decrease in time with $4\times$ increase in the number of places and the number of molecules. In comparison, *Custom* has around $2\times$ increase in time with $4\times$ increase in number of places.

Fig. 12 demonstrates the data scalability of *ADS* on large input data (number of molecules) varying from 8K to 32K on BG/P, with 3 threads (cores) per place and total 256 places. *ADS* incurs around $3.6\times$ increase in time with $4\times$ increase in the number of molecules. *Custom* has $1.5\times$ increase in time, with $4\times$ increase in the number of molecules.

Fig. 13 demonstrates the strong scalability of *ADS* for increasing number of cores, with up to 4 places on the Intel multi-core cluster. *ADS* achieves a *super-linear* of around $19.32\times$ with $6\times$ increase in the total number of cores; while *Custom* achieves $21\times$ speedup for the same increase in cores. The reason for the super-linear speedup for both *ADS* and *Custom* is that with smaller number of cores, all molecules are on a single place which hosts a single region for calculating forces between the molecules and hence more computations are performed. For larger number of cores, such as 24, 4 places are used, which means the molecules have spread around to 4 regions due to which the number of force calculations have gone down. Further, *ADS* performance is within 15.8% of the performance of *Custom* at 4 cores and within 26% at 24 cores.

Fig. 14 demonstrates the weak scalability of *ADS* with increasing number of total cores on the Intel multi-core cluster. *ADS* has only around $1.25\times$ increase in time with $3\times$ increase in number of cores and the input number of molecules (5K to 16K molecules). In comparison, *Custom* incurs around $2\times$ increase in time with $3\times$ increase in number of cores and molecules. The *ADS* performance is within 54% from the performance of *Custom* at 12 cores, even though initially this gap is around $2.5\times$.

Fig. 15 demonstrates the data scalability of *ADS* on large input data (number of molecules) varying from $4K$ to $32K$ on the Intel multi-core cluster, with 6 threads per place and total 4 places. *ADS* incurs around $15\times$ increase in time with $8\times$ increase in data. The reason for this increase is that as the number of molecules increases per region the number of force calculations increases more than linearly. The same pattern is observed for *Custom*, $68\times$ increase in time, with $8\times$ increase in the number of molecules. Further, *ADS* performance is within 50% of the performance of *Custom* even for large number of molecules, 32K. Note that this gap in performance is much lower at 16K molecules but suddenly jumps at 32K molecules due to increase in scheduling overheads.

5.2 Performance Comparison on Intel NUMA Architecture

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. 16 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

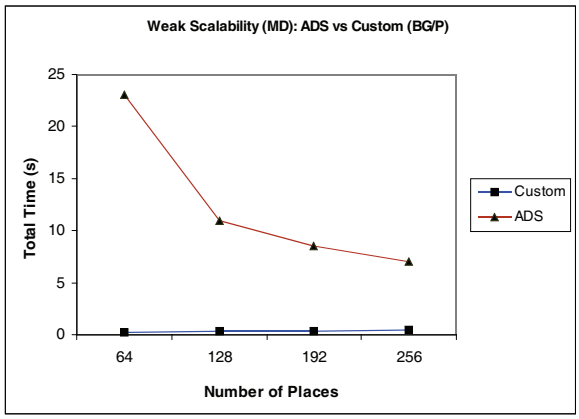

Figure 11. MD: Weak Scalability (BG/P)

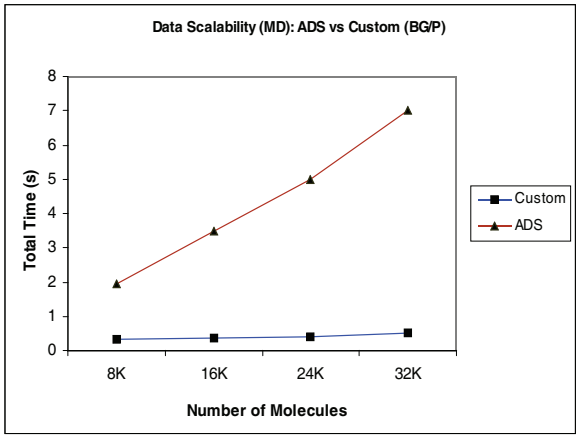

Figure 12. MD: Data Scalability (BG/P)

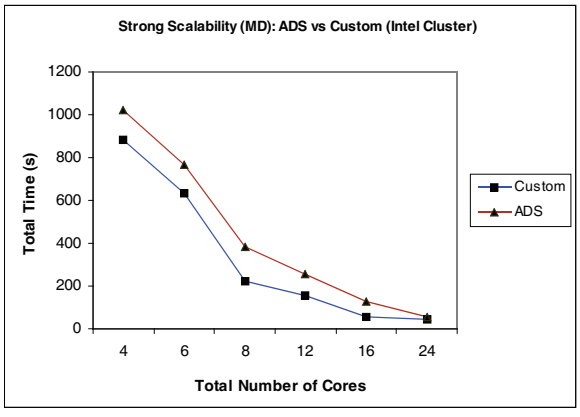

Figure 13. MD: Strong Scalability (Intel Cluster)

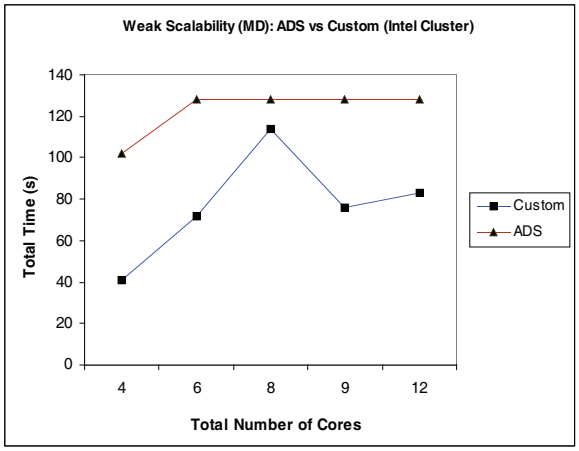

Figure 14. MD: Weak Scalability (Intel Cluster)

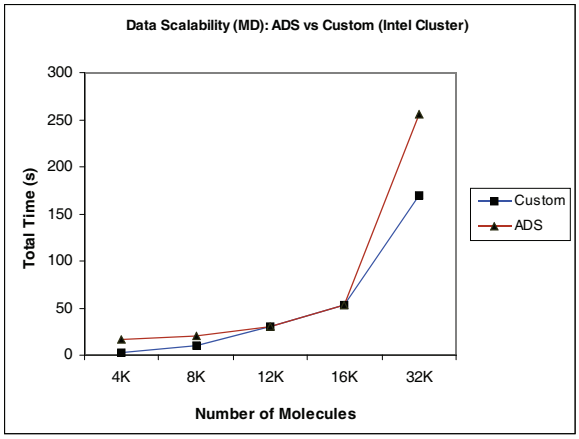

Figure 15. MD: Data Scalability (Intel Cluster)

around $1.9\times$ better than *CORG*, but later this gap stabilizes at around $1.20\times$.

5.3 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.

Fig. 17 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. 18 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

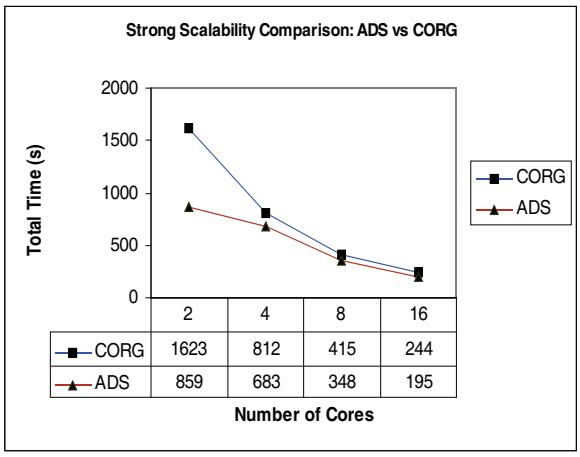

Figure 16. CORG vs ADS

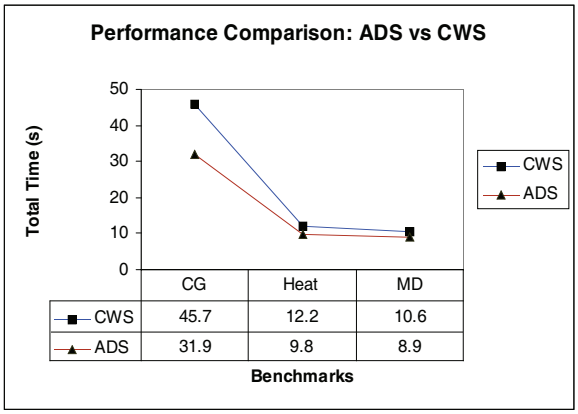

Figure 17. ADS vs CWS

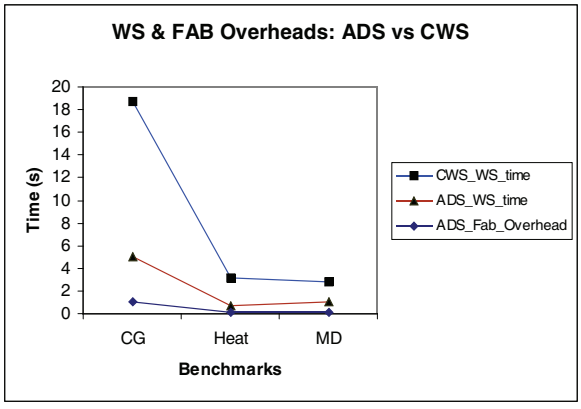

Figure 18. ADS vs CWS

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. 19 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. 20 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. 21 plots the work stealing time and *FAB* stealing time variation with changing multi-place configurations for the Heat 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. 22 gives the variation (for MD benchmark) 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

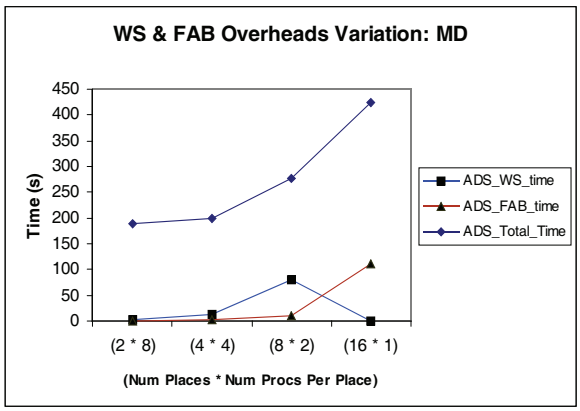

Figure 19. Overheads - MD

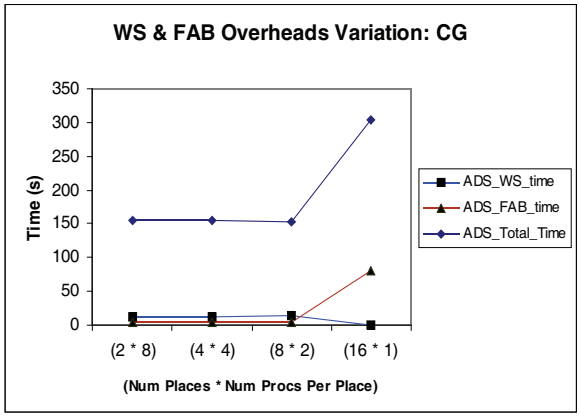

Figure 20. Overheads - CG

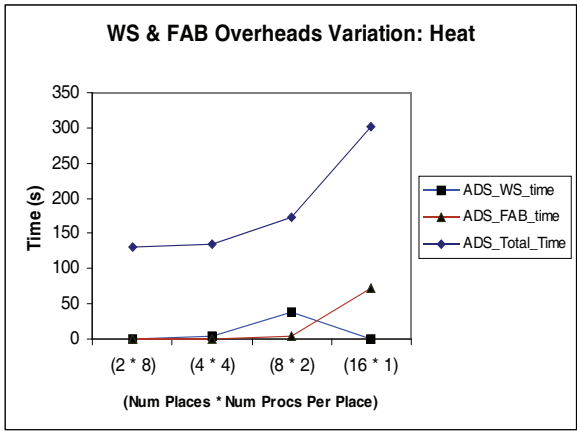

Figure 21. Overheads - HEAT

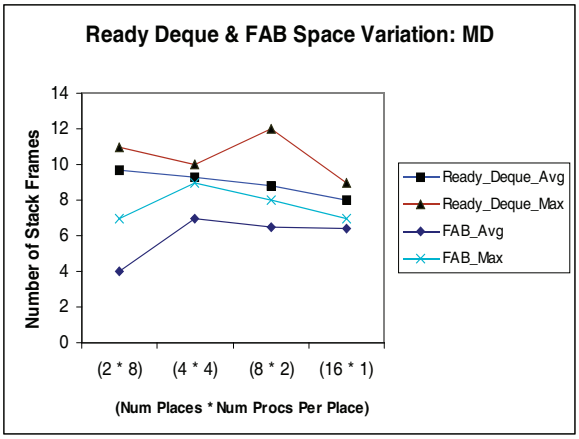

Figure 22. Space Util - MD

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. 24 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. 23 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 multi-objective affinity driven online distributed scheduling of parallel computations. We have provided detailed theoretical analysis of the time and message complexity bounds of our algorithm. On multi-core clusters including Blue Gene/P (MPP architecture) and Intel multi-core cluster, we have shown performance close to custom MPI+Pthreads code. Further, strong, weak and data scalability have been demonstrated in multi-core clusters. Using well known benchmarks, on Intel NUMA architecture, our algorithm demonstrates around 16% to 30% performance gain over typical Cilk style scheduling. Detailed experimental analysis illustrates the efficient space utilization as well. This is the

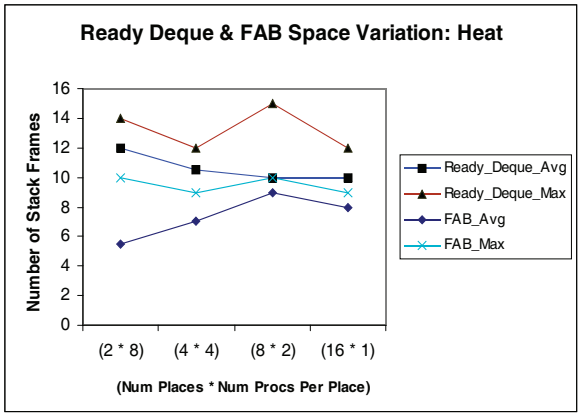

Figure 23. Space Util - HEAT

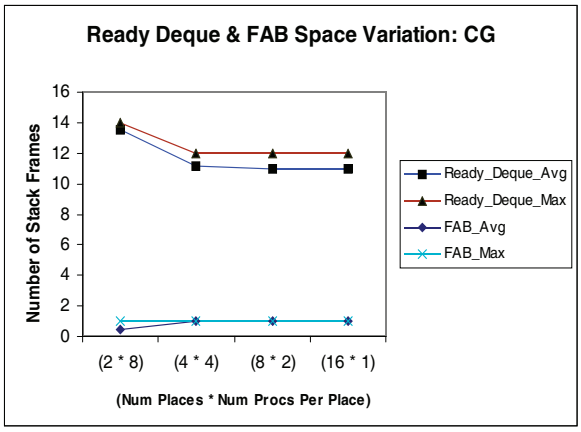

Figure 24. Space Util - CG

first such work for multi-objective affinity driven online distributed scheduling of parallel computations in a multi-place setup. In future, we plan to look into detailed space-time tradeoffs and markov-chain based modeling of the 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 specification 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] P. Berenbrink, T. Friedetzky, and L.A. Goldberg. A natural work-stealing algorithm is stable. In *Proceedings of the 42th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 178 – 187, 2001.

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

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

[8] 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.

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

[10] Exascale Study Group and Peter Kogge et.al. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, Sep 2008.

[11] A. Narang, A. Srivastava, Naga P.K. Katta, and R. K. Shyamasundar. Affinity driven distributed scheduling algorithm for parallel computations. In *ICDCN*, Bangalore, India, January 2011.

[12] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In *ISAAC (2)*, pages 291–302, 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.