Dynamic Distributed Scheduling Algorithm for State Space Search

Ankur Narang[1], Abhinav Srivastava[1], Ramnik Jain[1], and R.K. Shyamasundar[2]

[1] IBM India Research Laboratory, New Delhi
{annarang,abhin122,ramnjain}@in.ibm.com
[2] Tata Institute of Fundamental Research, Mumbai
shyam@tifr.res.in

Abstract. Petascale computing requires complex runtime systems that need to consider load balancing along with low time and message complexity for scheduling massive scale parallel computations. Simultaneous consideration of these objectives makes online distributed scheduling a very challenging problem. For state space search applications such as UTS, NQueens, Balanced Tree Search, SAT and others, the computations are highly irregular and data dependent. Here, prior scheduling approaches such as [16], [14], [7], HotSLAW [10], which are dominantly locality-aware work-stealing driven, could lead to low parallel efficiency and scalability along with potentially high stack memory usage.

In this paper we present a novel distributed scheduling algorithm (*LDSS*) for *multi-place*[1] parallel computations, that uses an unique combination of d-choice randomized remote (inter-place) spawns and topology-aware randomized remote work steals to reduce the overheads in the scheduler and dynamically maintain load balance across the compute nodes of the system. Our design was implemented using GASNet API[2] and POSIX threads. For the UTS (Unbalanced Tree Search) benchmark (using upto 4096 nodes of Blue Gene/P), we deliver the best parallel efficiency (92%) for $295B$ node binomial tree, better than [16] (87%) and demonstrate super-linear speedup on 1 Trillion node (largest studied so far) geometric tree along with higher tree node processing rate. We also deliver upto 40% better performance than Charm++. Further, our memory utilization is lower compared to *HotSLAW*. Moreover, for NQueens ($N = 18$), we demonstrate superior parallel efficiency (92%) as compared Charm++ (85%).

1 Introduction

State space search problems such as planning and scheduling problems in manufacturing industries and world wide web, VLSI design automation problems (routing, floorplanning, cell placement and others), N-Queens [8], Traveling Salesman problem and other discrete optimization problems are very fundamental in nature and hence frequently used in many industry application domains and systems research. Since all these problems are NP-Hard, one needs to resort to systematic but intelligent state

[1] 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.

[2] http://gasnet.cs.berkeley.edu

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 141–154, 2012.

space search to find optimum solutions. The states and the transition function(s) (including constraints) between the states are defined according to the nature of the state space search problem. The objective of the state space search problem is to find a path from a start state to a desired goal state (or a path from the start to each among a set of goal states). For a lot of state space search problems, in order to search the given state space, one constructs a *search tree* where each *node* in the search tree represents the state reached during the search.

Even though it seems that state space search problems require exponential number of processors (as compared to graph algorithms such as depth-first search etc.) since their worst case time is almost always exponential, the average time complexity of heuristic search algorithms for some problems is polynomial [17] [12]. Furthermore, there are heuristic search algorithms that find suboptimal solutions for specific problems in polynomial time. In such cases, bigger problem instances can be solved using large scale parallel computing infrastructure. Many discrete optimization problems (such as robot motion planning, speech understanding, and task scheduling) require realtime solutions. For these applications, parallel processing may be the only way to obtain acceptable performance. Since the state space search involves higher irregular computation DAG, it suffers from severe load balancing problems.

Further, with the advent of petascale machines such as K-Computer [3], Jaguar [4], Blue Gene/Q [5] and others, there is an imminent demand for strong performance and scalability of large scale computations along with improved programmer productivity. Thus, there is a strong need to have efficient scheduling frameworks as part of run-time systems that can meet these performance and productivity objectives simultaneously. For handling large parallel computations, the scheduling algorithm (in the run-time system) should be designed to work in a *distributed* fashion. For the execution of dynamically unfolding irregular and data-dependent parallel computations, the online scheduling framework has to make decisions dynamically on where (which place and core/processor) and when (order) to schedule the computations. Further, the critical path of the scheduled computation is dependent on load balancing across the cores as well as on the computation and communication overheads. The scheduler needs to maintain appropriate trade-offs between load balancing, communication overheads and space utilization. Simultaneous consideration involving space, time, message complexity and load balance makes distributed scheduling of large scale parallel state space search applications a very challenging problem.

Distributed Scheduling for parallel computations is a well studied problem in the shared memory context starting from the pioneering research by Blumofe and Leiserson [3] on Cilk scheduling, followed by later work including [2] [1] [4] [6] amongst many others. These efforts are primarily focused on work-stealing efficiency improvement in shared-memory architectures without considering explicit affinity annotations by the programmer. With the advent of distributed memory architectures, lot of recent research on distributed scheduling looks at multi-core and many-core clusters [16] [15]. All these recent efforts primarily achieve load balancing using (locality-aware) work

[3] http://www.fujitsu.com/global/about/tech/k/

[4] http://www.nccs.gov/computing-resources/jaguar/

[5] http://www-03.ibm.com/systems/deepcomputing/solutions/bluegene/

stealing across the nodes in the system. Although, this strategy works well for slightly irregular computation such as UTS for geometric tree, it could result in large parallel inefficiencies when the computation is highly irregular (binomial tree for UTS). Certain other approaches such as [14] consider limited control and no data-dependencies in the parallel computation, which limits the scope of applicability of the scheduling framework.

In this paper, we address the following distributed scheduling problem.

Given:
(a) A parallel computation DAG (Fig. $1(a)$) that represents a parallel multi-threaded computation. Each node in the DAG is a basic operation (instruction) such as and/or/add etc. Each edge in the DAG represents one of the following: (a) Spawn of a new thread; (b) Sequential flow of execution;or, (c) 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 2).
(b) A cluster of *n* SMPs (refer Fig. $1(b)$) as the target architecture on which to schedule the computation DAG. Each SMP 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:
(a) good trade-off between load-balance across the nodes and communication overheads;
(b) Low space, time and message complexity for execution.

In this paper, we present the design of a novel distributed scheduling algorithm (referred as *LDSS*) that **combines** topology-aware *inter-place prioritized random* work stealing with d*-choice based randomized distributed remote spawns* to provide automatic dynamic load balancing across places. Our *LDSS* algorithm partitions the compute nodes of the target system into *disjoint groups*. By using higher priority for limited radius (within a group) work stealing as well as remote spawns across the places (as compared to farther off, outside the group) our algorithm achieves low overheads. The remote spawns happen within the group to maintain affinity, while they are enabled across the groups to improve load-balance in the system. By controlling the **rate of remote spawns** [6], **rate of remote work steals**, **granularity of work steals** and **group size** one can obtain *a balanced trade-off* point between load balancing, scheduling overheads and space utilization. Our main contributions are as follows:

- We present a novel online distributed scheduling algorithm (referred to as *LDSS*) that uses an elegant *combination of topology-aware* remote (inter-place) spawns based on randomized d-choice load balancing and remote prioritized random work steals to reduce the overheads in the scheduler and to dynamically maintain load balance across the compute nodes of the system.
- By tuning the parameters such as granularity of remote steals, remote work-steal rate, value of d in d-choice based remote spawns, compute group-size and others we obtain optimal trade-offs between load-balance and scheduling overheads

[6] Ratio of remote spawned threads to total spawned threads at a processor.

that results in scalable performance. The *LDSS* algorithm was implemented using GASNet API and POSIX threads to enable *asynchronous communication* across the nodes and improve *computation-communication* and *communication-communication* overlap.

– Using upto 4096 nodes of Blue Gene/P we obtained superior performance as compared to prior approaches. For the binomial tree UTS (Unbalanced Tree Search) benchmark [7], LDSS delivers: (a) Upto around 40% better performance than Charm++ [15] and [16]; (b) Best parallel efficiency (92%) for $295B$ node tree as compared to best prior work [14] [16] (87%). LDSS demonstrates super-linear speedup for 1 Trillion node geometric tree and best processing rate of around 4GNodes/s for 16Trillion node geometric tree (largest studied so far by any prior work). Further on benchmarks such as NQueens [8], LDSS demonstrates superior parallel efficiency as compared to Charm++ on Blue Gene/P, MPP architecture.

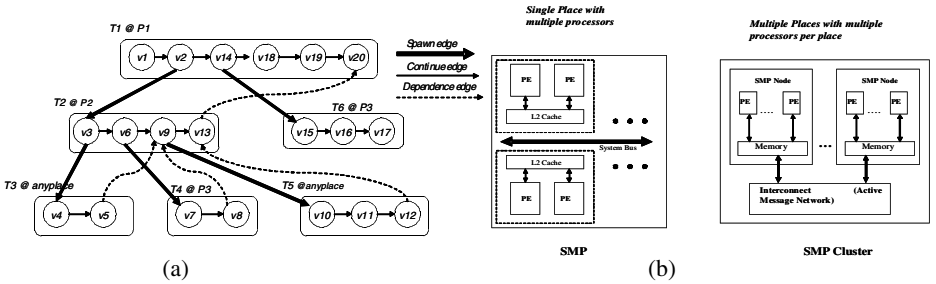


Fig. 1. (a) Computation DAG. (b) Multiple Places: Cluster of SMPs

2 System and Computation Model

The system on which the *computation DAG* is scheduled is assumed to be cluster of *SMPs* connected by an *Active Message Network* (Fig. $1(b)$). Each *SMP* is a group of processors with shared memory. Each SMP is also referred to as *place* in the paper. Active Messages *((AM)* [8] 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 DAG (Fig. $1(a)$) that either represent: (a) creation of new activities

[7] `http://barista.cse.ohio-state.edu/wiki/index.php/UTS`

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

(*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); and it may or may not have an affinity annotation defined by the programmer. Fig. 1 shows a strict computation dag where: $v1..v20$ denote nodes, $T1...T6$ are threads and $P1..P3$ denote places).

Based on the structure of dependencies between the nodes in the computation DAG, there can be multiple types of parallel computations such as: (a) ***Fully-strict computation***: Dependencies are only between the nodes of a thread and the nodes of its immediate parent thread; and, (b) ***Strict computation***: Dependencies are only between the nodes of a thread and the nodes of any of its ancestor threads.

3 LDSS: Scheduling Algorithm

Our distributed scheduling algorithm, *LDSS*, attempts to achieve communication efficient load balancing across the places with low scheduling overheads. In order to achieve this goal, we make the following design choices: (a) **Topology Awareness:** The places in the system are clustered into small *groups* based on their distances amongst each other in the topology of the underlying target architecture; (b) **Two-level Work Stealing:** Our algorithm uses work-stealing at two-levels. One is intra-place randomized work stealing to achieve load balance across the processors within a place. The other is inter-place prioritized (topology-aware) randomized work stealing that provides load balance across the places in the system; (c) **Load Balance driven Randomized Work Pushing:** LDSS incorporates (topology-aware) work-pushing across the places (nodes) in the system. This uses the d-choice randomized load balancing algorithm to achieve low load imbalance across the groups. The rate of such *remote spawns* is automatically adjusted during the algorithm;and, (d) **Dedicated Communication Processor:** In order to handle inter-place spawns we assign a dedicated communication processor in each node (place). This communication processor uses GASNet API to enable asynchronous communication and improves the performance of the scheduling algorithm by enabling *computation-communication* overlap as well as *communication-communication* overlap across the places. Within a place, the online unfolding of the computation DAG happens in a depth-first manner to enable efficient space and time execution. To achieve load balancing within a place, work-stealing is enabled to allow load-balanced execution of the computation sub-graph associated with that place. The computation DAG unfolds in an online fashion in a breadth-first manner across places when the threads are pushed (remote spawns) onto remote places for better load balance. This execution strategy leads to low overall stack space requirement as compared to prior approaches which use a combination of work-first and help-first policies [10].

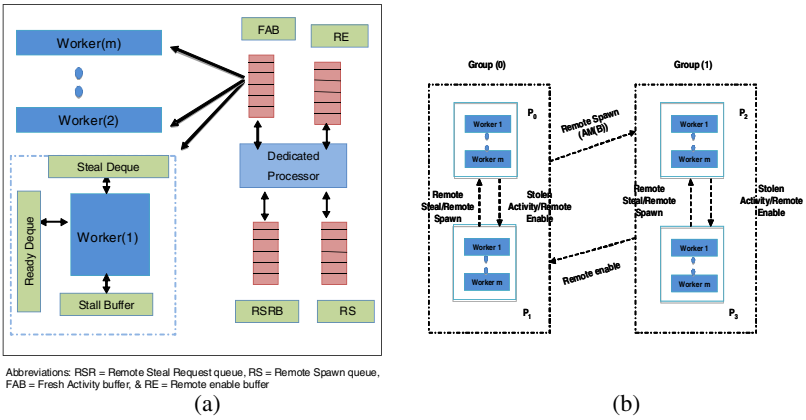


Fig. 2. (a) Distributed data-structures at a Place. (b) Work Stealing & Remote Spawns

3.1 Distributed Data Structures

Each place has a dedicated communication processor (different from workers) to manage remote communication with other places. This processor manages the following data-structures (Fig. 2(b)): (a) *Fresh Activity Buffer*(*FAB*) is a non-blocking FIFO data-structure. It contains threads that are remote spawned onto this place by remote nodes due to either the place annotation of the thread or the need for inter-group load balancing; (b) *Remote Spawn queue* (*RS*) is a non-blocking FIFO data-structure which contains all the threads that are to be remote spawned by local processors onto remote places; (c) *Remote Enable Buffer* (*RE*) is a non-blocking deque data structure, which contains all the remote enable signals issued by local processors to remote places;and, (d) *Remote Stealing queue* (*RSRQ*) is a FIFO data-structure contains all the remote steal requests received by this particular place from other places within the same group.

Each worker (processor) at a place has the following data-structures (refer Fig. 2(b)): (a) *Ready Deque*: is a deque that contains the threads of the parallel computation that are ready to execute locally. This is accessed by the local processor only; (b) *Steal queue*: is a non-blocking deque that contains threads that are ready to be stolen by the some other processor at a local or remote place. It is accessed by other local processors or communication processor for work stealing from this processor. In helps in reducing the synchronization overheads on the local processor;and, (c) *Stall Buffer*: is a deque that contains the threads that have been stalled due to dependency on another thread that are either spawned locally or remotely in the parallel computation. This is only accessed by the local processor.

3.2 Algorithm Design

During execution, the *LDSS* algorithm is able to keep track of data and control dependencies in the computation DAG, by using enable signals. Flow of enable signals across the places is managed by the dedicated communication processor at each place, using the *Remote Enable* buffer. The root place receives communication from each place on

the status of work at each place and based on the termination condition of the program, the root node sends termination signal to each node. This technique can be further enhanced using well-known global termination detection techniques. The actions taken by the (general) processors and the dedicated communication processor at each place, P_i, are described below.

General Processor Actions: At any step, a thread A at the r^{th} processor (at place i), W_i^r, may perform the following actions:

1. **Spawn**
   (a) A spawns B locally: B is successfully created and starts execution whereas A is pushed into the bottom of the *Ready Deque*.
   (b) A spawns B remotely: (i) If affinity for B is for a place, P_j, the target_place = P_j. (ii) Else, if B is anyplace thread, then determine the target_place using d-choice randomized selection. (iii) Active message for B is enqueued on head of the *Remote Spawn queue* with the destination as the target_place.
2. **Terminates** (A terminates): The processor 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 tries to transfer all the threads from *Steal queue* to *Ready Deque* and pick from the bottom of the deque. If *steal queue* is empty then it steals from the top of other processors' *Steal queue*. Each failed attempt to steal from another processor's *Steal queue* 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 processor is chosen from the same place. If no thread is available at that place, then enable inter place work stealing. (The communication processor helps in inter-place work-stealing by using the d-choice prioritized random selection of victim places and deciding the target_place.)
3. **Stalls** (A stalls): A thread may stall due to control or data 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 . If B is a local thread then the state of B changes to enabled and it is pushed onto the appropriate position of the *Ready Deque*. It B is remotely stalled then push the enable signal for that place at the bottom of the *Remote Enable* buffer.

Dedicated Communication Processor Actions: At any moment during the execution, the **dedicated communication processor** at place i will try pick up an active message from the bottom of the *Remote spawn queue*. Each failed attempt is followed by attempt to pick up an enable signal from bottom of the *Remote Enable* buffer. If there is no enable signal in *Remote Enable buffer* and inter place work stealing is enabled then it randomly non-uniformly (with priority) chooses d distinct places (with higher priority to places within its own *group*) and sends the active messages requesting workloads. On receiving reply from these places, it selects that target_place (*victim* place) as the one with the highest load (as measured in the prior time interval). If this fails, then it tries to pick up the request from the bottom of the *Remote Stealing* buffer and sends it an available thread at this place. All these operations require asynchronous and one-sided

communication with other places. Hence, we implemented our *LDSS* algorithm using *GASNet*.

The dedicated communication processor also helps in maximizing *computation-communication overlap* as well as *communication-communication overlap*. When, a thread needs data from another place, it sends the request to the communication processor. The communication processor forwards that request to the place that contains that data. This goes in parallel with the computation that can be performed at the processor where the data request originated. Hence, one gets computation-communication overlap. Moreover, multiple communication requests including remote steal requests, remote enable and d-choice selection all can proceed in parallel with different places, leading to effective communication-communication overlap.

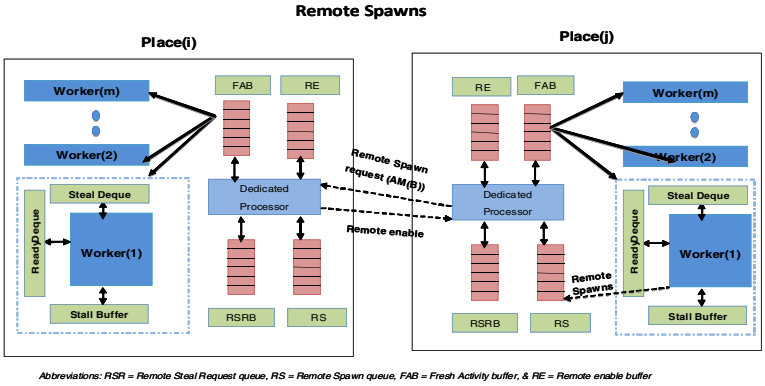


Fig. 3. Remote-spawn from Place(i) to Place(j)

Remote Spawns: Any processor that needs to spawn a thread (Fig 3), enqueues the active message for creation of that thread at the head of the *Remote Spawn* buffer. The dedicated communication processor pops the active message from the *Remote Spawn* buffer and sends it to the appropriate place asynchronously. The communication processor uses d-choice randomized load balancing for choosing the appropriate destination place. Here, random d groups are selected and the one with the lowest load is chosen as the destination. Each place maintains a load vector that contains load of d places. This load vector is updated (d each time) at periodic intervals. The ***rate of remote spawns*** is adjusted automatically (by considering relative load difference between this node and system average load) to reduce overheads while at the same time provide optimal trade-off between load balancing across the places and scheduling overheads. The d-choice based remote spawns result in good load balancing [11] while keeping low scheduling overheads. It is well-known [11] that pure random assignment of m balls (threads) to n bins ($m >> n$) leads to $O(\sqrt{\frac{m \log(n)}{n}})$ gap across the bins (servers, processors) while d-choice based assignment leads to $O(\ln \ln(n))$ gap across the bins. Thus, the instantaneous load-imbalance across the nodes (places) reduces in the system.

Workstealing: Each core (processor) uses *Ready Deque* (lockless queue for threads intended for local execution and *Steal queue* (synchronized queue) for threads that can be stolen. Each place in the system is associated with one and only one *group*. For inter-place work stealing higher priority is given to the groups close in the target topology as compared to the groups farther away. Once all the processors becomes idle at a place, the dedicated (communication) processor at that place (*thief*) queries a randomly selected place (*victim*) about its load. When the *victim* dedicated communication processor receives the request for worksteal from a *thief*, it randomly deques a thread from one of its local processors' *Ready deque* and sends it to thief place for its continuation.

4 Results and Analysis

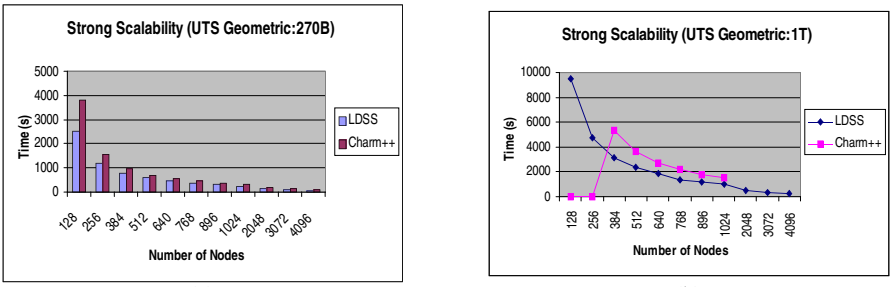
Experimental Setup: We used upto 4096 compute nodes/places (with 4 cores/ processors per place, total 16384 cores in the system) of Blue Gene/P (*Watson 4P* [9]) for empirical evaluation of our distributed scheduling algorithm. Each compute node (place) in *Watson 4P* is a quad-core chip with frequency of 850 MHz having 4 GB of DRAM and 32 KB of L1 instruction and data 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. For efficient compute and communication overlap, *GASNet* was used since it provides asynchronous one sided message passing primitives. GASNet is a language-independent, low level networking layer that provides network-independent, high performance communication primitives tailored for implementing Parallel Global Address Space. GASNet's *DCMF* (Deep Computing Messaging Framework) conduit is the native port of GASNet to BlueGene/P architecture as it uses DCMF for the lower level communication between nodes.

Benchmarks: We implemented our distributed scheduling algorithm (*LDSS*) using pthreads (NPTL API) and GASNet as the underlying communication layer. The LDSS algorithm and benchmarks were compiled using *mpixlc_r* with optimization options *-O3*, *-qarch=450*, *-qtune=450* and *-qthreaded*. We present comparison of performance and scalability with Charm++ [15] on Blue Gene/P architecture and show that we have superior results. The benchmarks used for evaluation include:

- **Unbalanced Tree Search** (UTS): The Unbalanced Tree Search problem is to count the number of nodes in an implicitly constructed tree that is parameterized in shape, depth, size and imbalance, and,
- **NQueens**: NQueens is a backtracking search problem to place N queens on a N by N chess board so that they do not attack each other. We target at finding all solutions for N Queen problem.
  Note: UTS and NQueens are *strict* parallel computations as both of them have parent-child dependencies.

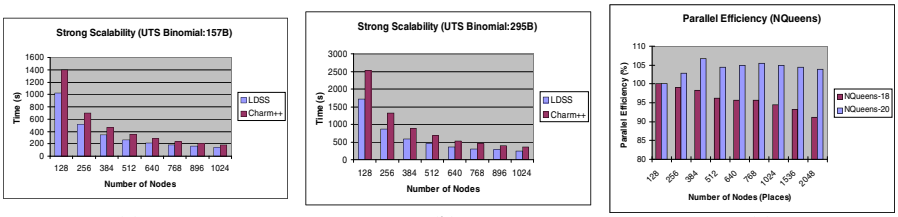
Scalability Analysis: Here, we present scalability analysis for the benchmarks. Fig. $4(a)$ and Fig. $4(b)$ demonstrate the strong scalability of *LDSS* algorithm with increasing

[9] `http://www.research.ibm.com/bluegene`



(a) (b)

Fig. 4. UTS: (a) Strong Scalability (270B) Geometric Tree. (b) Strong Scalability (1T) Geometric Tree



(a) (b) (c)

Fig. 5. UTS: (a) Strong Scalability (157B) Binomial Tree. (b) Strong Scalability (295B) Binomial Tree (c) Strong Scalability (NQueens).

number of nodes (places) from 128 to 4096 for geometric type UTS tree with 270 Billion, and 1 Trillion tree size respectively. Here, the granularity of work-stealing was kept as 50 and the group size chosen was 8. For 270B geometric tree, the *LDSS* algorithm achieves super-linear speedup of around $36.6\times$ (from $2524s$ to $69s$) with $32\times$ increase in number of nodes. The *LDSS* algorithm has better performance as compared to Charm++ by at least 28% throughout the variation in compute nodes. For the 1T geometric tree, the *LDSS* algorithm also achieves super-linear speedup of around $37.5\times$ (from $9491s$ to $253s$) with $32\times$ increase in number of nodes. Here again, the *LDSS* algorithm has better performance ($2352s$) as compared to Charm++ ($3667s$) by around 36% at 512 compute nodes, and by 35% ($1007s$ vs $1541s$) at 1024 nodes. Charm++ gave memory error for 128 and 256 nodes (places).

On 4096 nodes, *LDSS* had a completion time of $69s$ for 270B nodes, $253s$ for 1T nodes, $993s$ for 4T nodes and $4037s$ for 16T nodes; which demonstrates better than linear ***data scalability***. For 16Trillion nodes, LDSS delivers processing rate of 3.96G Nodes/s, which is the best reported so far in the literature. Further, the parallel efficiency achieved is slightly better as compared to the best prior work [7], and this is demonstrated on the geometric tree 16Trillion tree nodes which is largest amongst the maximum sizes considered by any prior work including [14] [16] [10].

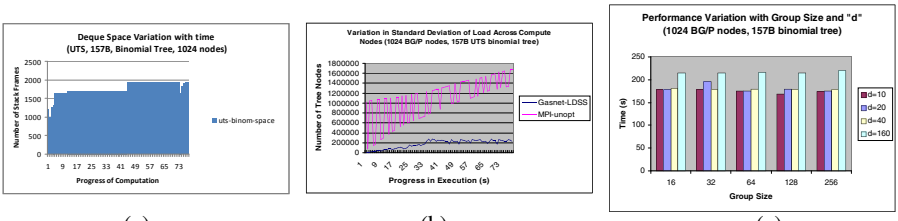
Fig. $5(a)$ and Fig. $5(b)$ demonstrate the strong scalability of *LDSS* algorithm with increasing number of nodes (places) from 128 to 1024 for binomial type UTS tree with 157 Billion, and 295 Billion tree size respectively. Here, the granularity of work-stealing was kept as 50, the group size chosen was 8, the base remote spawn rate was set at 50

and d was chosen as 3 for d-choice randomized load balancing during remote spawns. For 157B binomial tree, the *LDSS* algorithm achieves a speedup of around $7.34\times$ (from $1021s$ to $139s$) with $8\times$ increase in number of nodes, resulting in parallel efficiency of around 92%. The performance of *LDSS* is better than Charm++ by around 27% at 128 nodes and by around 22% at 1024 nodes.

For the 295B binomial tree, the *LDSS* algorithm also achieves a speedup of around $7.34\times$ (from $1715s$ to $234s$) with $8\times$ increase in number of nodes, resulting in parallel efficiency of 91.75%. The efficiency achieved is better than the best prior work [16] by around 5%. Further, for $295B$ nodes, *LDSS* has lower time than Charm++ by around 32% at 128 nodes and by around 35% at 1024 nodes.

The efficiency for binomial tree is lower than the geometric tree case since the binomial tree has larger depth and smaller breadth and hence more unbalanced as compared to the geometric tree. Due to this, the scheduling algorithm incurs larger overheads of remote spawns and work stealing in-order to achieve load balance across the compute nodes in the system. Hence, the geometric tree is able to achieve high efficiency even without remote spawns. The average (across varying number of compute nodes) single node performance of *LDSS* for Binomial tree is $1.1M$ nodes/sec as compared to $0.85M$ nodes/s for Charm++; while that for Geometric tree it is $0.96M$ nodes/s (for *LDSS*) as compared to $0.70M$ nodes/s for Charm++.

The parallel efficiency results (w.r.t. 128 nodes (places)) for NQueens benchmark are presented in Fig. $5(c)$. While for NQueens 20, LDSS delivers super-linear scalability and sustains parallel efficiency of 103% even at 2048 nodes; for NQueens 18 the parallel efficiency drops to 91% at 2048 nodes (places). This is due to exponential increase in size of NQueens 20 w.r.t. NQueens 18. For NQueens 18, the parallel efficiency achieved by LDSS is better than that for Charm++ (around 85%) [15].



(a) (b) (c)

Fig. 6. UTS: (a) Space Usage (UTS). (b) Standard deviation of Load Across Compute Nodes (UTS) (c) Performance Variation with Group Size and d value.

Fig. $6(c)$ illustrates the impact of change in group size on the overall performance of the *LDSS* algorithm. This variation is considered with different values of d in d-choice randomized spawns. For most group sizes (16, 64, 128, 256), as the value of d increases, the time increases while for group size 32, d value equal to 40 gives the best performance. Thus, there is an optimal combination of group size and d that gives the best performance. In general, as d increases for a given group size, the communication overheads increase leading to a larger overall time, while for some values of group size, larger d could also lead to better load balance within the system. For $d = 32$

we observed this by studying the load across the compute nodes in the system. This represents complicated trade-offs between load balance and communication overheads involved in scheduling binomial UTS trees.

Fig. $6(a)$ presents space usage (in number of stack frames) of the total space usage per processor (including the space used by the dedicated communication processor) as the computation progresses in the case of UTS/binomial tree with $157B$ nodes. The maximum space used is less than 2000 stack frames. This at least $3\times$ lesser than that reported by HotSLAW [10], which reports maximum space usage of around 8000 stack frames and stays above 2000 stack frames for quite sometime. This is because LDSS does not use help-first policy which leads to BFS expansion of the graph and larger usage of space, while HotSLAW uses a combination of work-first and help-first policy as does SLAW [6]. Fig. $6(b)$ reports the standard deviation in load across the compute nodes in the system for LDSS with tuned (*gasnet-opt*) and untuned (*MPI-unopt*) parameters [10], as the computation progresses for $157B$ binomial tree. By using GASNET and parameter tuning LDSS achieves around $8\times$ lower standard deviation (and hence better load balance) as compared to MPI implementation and untuned parameters.

5 Related Work

Distributed Scheduling for parallel computations is a well studied problem in the shared memory context starting from the pioneering research by Blumofe and Leiserson [3] on Cilk scheduling, followed by later work including [2] [1] [4] [6] amongst many others. These efforts are primarily focused on work-stealing efficiency improvement in shared-memory architectures without considering explicit affinity annotations by the programmer. With the advent of distributed memory architectures, lot of recent research on distributed scheduling looks at multi-core and many-core clusters.

Olivier et.al. [14] consider work stealing algorithms in distributed and shared memory environments, with one sided asynchronous communications. This work considers task migration on pull based mechanism and ignores affinity as well as it considers computations with no dependencies.

Dinan et.al. [7] construct distributed and local task pools for its dynamic load balancing model. [7] restricts the execution model by requiring that all tasks enqueued in task pool are independent. The model is confined to tasks that require only parent-child dependencies not other way around. Our model supports all the computations that are *strict* in nature hence allowing tasks to wait for completion of other tasks.

Saraswat et.al. [16] introduce a lifeline based global load balancing technique in *X10* which provides better load balancing for tasks as compared to random work stealing, along with global termination detection using the *finish* (X10) construct. Our algorithm considers multiple workers per place and handles data dependencies across the threads in the computation tree. We demonstrate better efficiency and performance on the binomial tree in UTS benchmark than [16].

Ravichandran et.al. [9] introduce work stealing for multi-core HPC clusters which allow multiple workers per place and two separate queues for local threads and for remote stealing, but this does not consider locality or data dependencies. Min et.al. [10]

[10] Rate of remote spawns, rate of workstealing, granularity of workstealing and group size.

present a task library, called *HotSLAW*, that uses *Hierarchical Victim Selection* (HVS) and *Hierarchical Chunk Selection* (HCS) to improve performance as compared to prior approaches. Our *LDSS* algorithm uses an elegant combination of two-level work stealing and remote-place (inter-group) work pushing to achieve optimal trade-offs between load balancing and scheduling overheads. Further, our space requirement is much lower than that reported by [10] for the UTS benchmark as presented in the Results section 4. Frameworks such as Scioto framework [5] and KAAPI [11] consider distributed setup but have not demonstrated results at large scale for state space search problems.

Charm++ is a C++ based parallel programming system that implements a message-driven migratable objects programming model, supported by an adaptive runtime system and work stealing [13] [15]. Charm++ supports work stealing across places [15] and uses a hierarchical mechanism [18] to migrate objects to places (processors) for load balancing. Zheng et.al. [18] consider hierarchical load balancing in Charm++. Our algorithm incorporates randomized d-choice based work pushing and prioritized inter-place work-stealing to ensure better instantaneous load balance across the places in the system. Further, on the *UTS* benchmark we demonstrate upto 40% better performance as compared to Charm++ on Blue Gene/P.

6 Conclusions and Future Work

We have addressed the challenging problem of online distributed scheduling of state space search oriented parallel computations, using a novel combination of d-choice based randomized remote spawns and topology-aware work stealing. On multi-core clusters such as Blue Gene/P (MPP architecture), our *LDSS* algorithm demonstrates superior performance and scalability (for UTS) and parallel efficiency (for NQueens benchmark) as compared to prior state-of-the-art approaches such as Charm++. For *UTS* (binomial tree) we have delivered highest parallel efficiency (close to 92%) for binomial tree (better than [16] which delivers 87%); and upto 40% better performance as compared to Charm++ [15]. In future, we plan to look into balanced allocation [11] based arguments to compute optimum trade-offs between work sharing and work stealing in large scale distributed environments.

References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: SPAA, New York, NY, USA, pp. 1–12 (December 2000)
2. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: SPAA, Puerto Vallarta, Mexico, pp. 119–129 (1998)
3. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)
4. Blumofe, R.D., Lisiecki, P.A.: Adaptive and reliable parallel computing on networks of workstations. In: USENIX Annual Technical Conference, Anaheim, California (1997)
5. Dinan, J., Krishnamoorthy, S., Larkins, D.B., Nieplocha, J., Sadayappan, P.: A framework for global-view task parallelism. In: Proceedings of the 37th Intl. Conference on Parallel Processing, ICPP (2008)

[11] `https://gforge.inria.fr/projects/kaapi/`

6. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: Slaw: A scalable localityaware adaptive work-stealing scheduler. In: IPDPS, pp. 1–12 (2010)
7. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: SC, Oregon, USA (November 2009)
8. Kalé, L.: An almost perfect heuristic or the n-queens problem. Information Processing Letters 34, 173–178 (1990)
9. Ravichandran, K., Lee, S., Pande, S.: Work Stealing for Multi-core HPC Clusters. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 205–217. Springer, Heidelberg (2011)
10. Min, S.-J., Iancu, C., Yelick, K.: Hierarchical work stealing on manycore clusters. In: PGAS 2011: Fifth Conference on Partitioned Global Address Space Programming Models. ACM (October 2011)
11. Mitzenmacher, M.: The Power of Two Choices in Randomized Load Balancing. PhD in computer science. Harvard University (1991)
12. Pearl, J.: Heuristics-Intelligent Search Strategies for Computer Problem Solving. Addison Wesley (1984)
13. Shu, W., Kale, L.: A dynamic scheduling strategy for the chare-kernel system. In: ACM/IEEE Conference on Supercomputing (Supercomputing 1989), New York, USA, pp. 389–398 (1989)
14. Oliver, S., Prins, J.: Scalable dynamic load balancing using UPC. In: ICPP, Oregon, USA (September 2008)
15. Sun, Y., Zheng, G., Jetley, P., Kale, L.V.: An adaptive framework for large-scale state space search. In: IPDPS Workshop, Alaska, USA, pp. 1798–1805 (2011)
16. Saraswat, V., Grove, D., Kambadur, P., Kodali, S., Krisnamoorthy, S.: Lifeline based global load balancing. In: PPOPP, Texas, USA (February 2011)
17. Wah, B.W., Li, G., Yu, C.F.: Multiprocessing of combinatorial search problems. IEEE Computer 18, 93–108 (1985)
18. Zheng, G., Meneses, E., Bhatele, A., Kale, L.V.: Hierarchical load balancing for charm++ applications on large supercomputers. In: 39th International Conference on Parallel Processing Workshops, ICPPW, pp. 436–444. IEEE Computer Society, Washington, DC (2010)