Stratified Sampling for Even Workload Partitioning Applied to IDA* and Delaunay Algorithms

Jeeva Paudel
Department of Computing Science
University of Alberta
jeeva@cs.ualberta.ca

Levi H. S. Lelis
Departamento de Informática
Universidade Federal de Viçosa
levi.lelis@ufv.br

José Nelson Amaral
Department of Computing Science
University of Alberta
amaral@cs.ualberta.ca

***Abstract*—This work presents Workload Partitioning and Scheduling (`WPS`), a novel algorithm for evenly partitioning the computational workload of large implicitly-defined work-list-based applications on distributed/shared-memory systems. In `WPS`, a stratified sampling technique estimates the number of work items that will be processed in each step of the target application. Then `WPS` uses this estimation to evenly partition and distribute the computational workload. An empirical evaluation on large applications — Iterative-Deepening A* (IDA*) applied to (4×4)- and (5×5)-Sliding-Tile Puzzles, Delaunay Mesh Generation, and Delaunay Mesh Refinement — shows that `WPS` is applicable to a range of applications. A coordination between `WPS` and existing work-stealing schedulers for intra-node load balancing yields additional speedups in the range of 18% to 40% compared to that achieved with the existing work-stealing schedulers alone. Such a coordination also outperforms an existing workload-partitioning scheme intended specifically for IDA* algorithms by 17% to 36%.**

I. Introduction

State-space exploration algorithms, such as A* [8] and IDA* [11], are fundamental algorithms in areas such as Artificial Intelligence and Combinatorial Optimization [21]. A state-space exploration problem consists of a start state and a transition function that receives a state and returns a set of child states. State-space exploration algorithms can be applied to search for a path to a goal state, or to a specific state that minimizes an objective function.

One way to achieve high performance in state-space exploration algorithms is to process disjoint portions of states in different processing nodes of a computer cluster. However, it is usually difficult to foresee how many states will be processed to solve a given problem because state-space exploration algorithms usually operate on implicitly-defined state spaces. When an implicit definition of the state space is used, the current state produces a list of child states that recursively produce other child states as the algorithm progresses. Such algorithms also use enhancements to reduce the effort of state-space exploration by avoiding processing of states deemed as unfruitful — *e.g.*, usage of heuristic functions to guide search. The difficulty in predicting the number of states processed by such algorithms in each region of the state space leads to potential load imbalances in parallel-processing solutions.

Mainstream programming systems employ *work-stealing* to alleviate load imbalance. Under work stealing, a processor that runs out of work steals work from another processor. Work stealing can reduce an application's execution time by making work available to idle processing nodes. However, it can also impose significant overheads of synchronization on shared data structures, and of communication across the network.

A. Contributions

We propose a technique for workload distribution that minimizes the need for expensive load-balancing operations such as work stealing. The technique, called Workload Partitioning and Scheduling (`WPS`), operates in four steps: first, it samples a small portion of the states to be processed using the statistical technique of stratified sampling [4]; then, based on such a sampling, it estimates the total number of states to be processed; in the third step, it partitions the states into M parts of sizes as equal as possible, where M is the number of processing nodes in the computer cluster; and finally, it distributes the states amongst M processing nodes.

`WPS` is general and can be applied to different problems. As a demonstration, we use `WPS` to parallelize three algorithms from different applications: (i) Iterative-Deepening A* (IDA*) [12] for finding least-cost paths in state-space search problems; (ii) an algorithm for generating Delaunay Meshes [2], [27]; and (iii) an algorithm for refining Delaunay Meshes [13]. These applications were chosen because they represent a large set of important problems, including path-finding, computational geometry, and combinatorial optimization problems. Our empirical evaluation indicates that `WPS` achieves its goal of minimizing the need for expensive load-balancing operations. When operating in tandem with an intra-node work-stealing scheduler, `WPS` is faster than traditional work-stealing schemes alone in the range of 18% to 40% .

II. Preliminaries

This paper refers to a state in an application's state-space as a work-item, or simply an item. Let $S(n^*) = (N, E)$ be a *Work-Item Tree* (*WIT*) rooted at item n^*, representing the set of items processed by an exploration algorithm while solving n^*. N is a set of items and E is the set of edges in the tree. A *WIT* is formed by the items reachable from n^*. For each $n \in N$, $child(n)$ is the set of items generated when n is processed: $child(n) = \{n_i | (n, n_i) \in E\}$. We call $child(n)$ the children of n, and we call the edges $(n, n_i) \in E$ the actions available from

2	8	3
1	6	4
7		5

	1	2
3	4	5
6	7	8

Fig. 1: A random state (left), and the goal state (right) of the (3×3)-Sliding Tile Puzzle, also known as the 8-puzzle.

n. In contrast with the Artificial Intelligence literature, a *node* refers to a processing node in a cluster, and not to a vertex in the *WIT*. Also, S is used to refer to $S(n^*)$ whenever n^* is clear from context. An item n is expanded when a computer node processes n. This work deals with implicitly-defined *WITs*, as described next.

Figure 1 shows the (3×3)-Sliding Tile Puzzle (8-puzzle), an example of a state-space exploration problem. For each state of this puzzle, there is a set of available actions. For instance, the state shown in Figure 1 (left) has three available actions: move tile 5, 6, or 7 onto the blank space. Moving a tile onto the empty space generates another state. The objective in this puzzle is to find the shortest sequence of actions that transform the given state to the goal state shown in Figure 1 (right).

The *WIT* of the 8-puzzle (as the other *WITs* we deal with in this paper) is implicitly defined. That is, the *WIT* is not available a priori, but items in the *WIT* can be generated by applying actions from the initial item. *WIT*s of the 8-puzzle have at most 181,440 different states, which could be stored explicitly in memory. However, we are interested in problems that are too large to be stored explicitly in memory. For example, one of the application domains we use is the (4×4)-Sliding-Tile Puzzle, which has $\frac{16!}{2}$ different states.

III. Problem Formulation

Given M processing nodes in a computer cluster and an implicitly defined *WIT*, the *Work-Load Distribution Problem* consists in partitioning the items in the *WIT* into M parts $W_1, W_2, \cdots, W_M$ of similar size. Our goal is to minimize $\sum_{i,j \in \{1,\cdots,M\}} |W_i| - |W_j|$, where $|W_i|$ is the size of W_i. In this paper all items in the *WIT* take approximately the same amount of time to be processed. However, the algorithm we introduce in this paper could be easily adapted to deal with items that have different processing times.

In addition to being implicitly-defined, the tree representing the *WIT* is often unbalanced, which poses a significant challenge for an even workload partitioning. The *WIT* is usually unbalanced because exploration algorithms use enhancements, such as a heuristic function [21], to guide the exploration to more promising parts of the state space (details in Section VIII-A). As a result, the tree will grow more quickly toward the directions deemed as promising by the algorithm.

Consider, for example, that an initial item produces two items i_1 and i_2, and that our cluster has two processing nodes. Given that the *WIT* is unbalanced, the size of the subtree rooted at i_1 might be very different from the size of the subtree rooted at i_2. A trivial solution of assigning i_1's subtree to one node and i_2's subtree to another node will lead to workload imbalance. The lack of *a priori* information about the sizes of the subtrees rooted at i_1 and i_2 further complicates the Work-Load Distribution Problem.

The `WPS` algorithm presented in this paper uses the statistical technique of stratified sampling introduced by Chen [4] for quickly partitioning the *WIT* into parts of similar size.

IV. Chen's Stratified Sampling

Knuth [10] presents a technique to estimate the size of the tree expanded by a search algorithm such as chronological backtracking. His technique repeatedly performs a random walk from the root of the tree. When all branches have the same structure, a random walk down one branch is enough to estimate the size of the entire tree. Knuth observed that his technique was not effective when the tree is imbalanced. Chen [4] addressed this problem by stratifying the search tree to reduce the variance of the sampling process. This paper refers to Chen's technique as *Stratified Sampling* (`StraSa`). `WPS` uses `StraSa` to estimate the *WIT* size and, based on such an estimation, it finds a partition of the items in the *WIT*.

Definition 1 (Stratification)**.** *Let $S = (N, E)$ be a* WIT. *$T = \{t_1, \ldots, t_n\}$ is a stratification for S if it is a disjoint partitioning of N. If $n \in N$, $t_i \in T$ and $n \in t_i$, then $T(n) = t_i$ states that the stratum of n is t_i.*

`StraSa` is a general approach for approximating any function of the form $\varphi(n^*) = \sum_{n \in S(n^*)} z(n)$, where $S(n^*)$ is a *WIT* rooted at n^* and z is any function assigning a numerical value to an item. $\varphi(n^*)$ represents a numerical property of the search tree rooted at n^*. For instance, if $z(n) = 1$ for all $n \in S(n^*)$, then $\varphi(n^*)$ is the size of the *WIT*. Instead of traversing the entire *WIT* and summing all z-values, `StraSa` assumes that subtrees rooted at items of the same stratum have equal values of φ and thus only one item of each stratum, chosen randomly, is expanded. This selective expansion is the key to `StraSa`'s efficiency because trees of practical interest are too large to be examined exhaustively.

Given an item n^* and a stratification T, `StraSa` estimates $\varphi(n^*)$ as follows. First, it samples the *WIT* rooted at n^* and returns a set A of *representative-weight* pairs, with one such pair for every unique stratum seen during sampling. Given a pair $\langle n, w \rangle \in A$ for stratum $t \in T$, n is the unique item of stratum t that was expanded during sampling and w is an estimate of the number of items of stratum t in the *WIT* rooted at n^*. $\varphi(n^*)$ is then approximated by $\hat{\varphi}(n^*)$, defined as

$$\hat{\varphi}(n^*) = \sum_{\langle n, w \rangle \in A} w \cdot z(n) \,. \quad (1)$$

Algorithm 1 shows `StraSa` in detail. The set A is divided into subsets, one for every layer in the search tree; $A[i]$ is the set of representative-weight pairs for the strata encountered at level i. In `StraSa`, the strata must be partially ordered such that an item's stratum is strictly greater than that of its parent in the *WIT*. Chen suggests that this constraint can always be guaranteed by adding the depth of an item in the *WIT* to the stratification and then sorting the strata lexicographically. In this implementation of `StraSa` the depth of exploration is

Algorithm 1: `StraSa`, a single probe

Input: root n^* of a tree and a stratification T
Output: a sampled tree ST represented by an array of sets A, where $A[i]$ is the set of pairs $\langle n, w\rangle$ for the items n expanded at level i, and an array of sets C, where $C[i]$ is the set of items n generated at level i but not expanded.

1: $A[0] \leftarrow \{\langle n^*, 1\rangle\}$
2: $i \leftarrow 0$
3: **while stopping condition** is false **do**
4: **for** each element $\langle n, w\rangle$ in $A[i]$ **do**
5: **for** each child $\hat{n}$ of n **do**
6: **if** $A[i+1]$ contains an element $\langle n', w'\rangle$ with $T(n') = T(\hat{n})$ **then**
7: $w' \leftarrow w' + w$
8: with probability w/w', replace $\langle n', w'\rangle$ in $A[i+1]$ by $\langle \hat{n}, w'\rangle$ and insert n' in $C[i+1]$; insert $\hat{n}$ in $C[i+1]$ otherwise
9: **else**
10: insert new element $\langle \hat{n}, w\rangle$ in $A[i+1]$
11: **end if**
12: **end for**
13: **end for**
14: $i \leftarrow i + 1$
15: **end while**

implicitly added to the stratification: strata at each tree level are treated separately by the division of A into the $A[i]$. If the same stratum occurs on different levels, the occurrences are treated as though they were of different stratum.

$A[0]$ is initialized to contain only the root of the *WIT* to be probed, with weight 1 (line 1). In each iteration (lines 4 – 13), all the items from $A[i]$ are expanded to get representative items for $A[i+1]$ as follows. Every item in $A[i]$ is expanded and its children are considered for inclusion in $A[i+1]$. If a child $\hat{n}$ has a stratum t that is already represented in $A[i+1]$ by another item n', then a *merge* action on $\hat{n}$ and n' is performed. A merge action increases the weight in the corresponding representative-weight pair of stratum t by the weight $w(n)$ of $\hat{n}$'s parent n (from level i) since there were $w(n)$ items at level i that are assumed to have children of stratum t at level $i+1$. $\hat{n}$ will replace the n' according to the probability shown in line 8. Chen [4] proved that this probability reduces the variance of the estimation. Once all the states in $A[i]$ are expanded, `StraSa` expands the items in $A[i+1]$. This process continues until it reaches a level i^* where $A[i^*]$ is empty.

One run of the `StraSa` algorithm is called a *probe*. $\hat{\varphi}^{(p)}(n^*)$ is the p-th probing result of `StraSa`. `StraSa` is unbiased, i.e., the average of the $\hat{\varphi}(n^*)$-values converges to $\varphi(n^*)$ in the limit as the number of probes goes to infinity. Chen [4] states the following theorem:

Theorem 1. *Given a stratification T and a set of p independent probes $\hat{\varphi}^{(1)}(n^*), \cdots, \hat{\varphi}^{(p)}(n^*)$ from a* WIT *$S(n^*)$, $\frac{1}{p}\sum_{j=1}^{p} \hat{\varphi}^{(j)}(n^*)$ converges to $\varphi(S)$ as p grows large.*

Each `StraSa` probe outputs a subtree of the *WIT* called *sampled tree* (ST). In constrast with Chen's version of `StraSa`, our version of the algorithm also outputs an array of sets C containing the items encountered during sampling which were not expanded. C is organized by levels, e.g., $C[i]$ is the set of items `StraSa` encountered but did not expand at level i of the *WIT*. The algorithm introduced in this paper uses C to evenly divide the workload among different processing nodes, as we explain next.

Algorithm 2: Workload Partitioning and Scheduling

Input: starting item n^* of the *WIT* and a stratification T
Output: solution for the problem represented by n^*

1: $[A, C] \leftarrow StraSa(n^*, T)$ // see Algorithm 1
2: $\chi \leftarrow ComputeSubtreeSizes(A, T)$ // see Algorithm 3
3: $\{W_1, W_2, \cdots, W_M\} \leftarrow BLDM(\chi, C)$ // see [16]
4: **for** $i \in \{1, \cdots, M\}$ **do**
5: asynchronously copy W_i to node i
6: **end for**

V. `WPS`: Workload Partitioning & Scheduling

Algorithm 2 shows a high-level description of `WPS`. `WPS` operates in four phases: sampling, estimating, partitioning, and distributing. First, we describe the four phases `WPS` when using a single `StraSa` probe, then in Section VI we describe how the algorithm employs multiple probes of `StraSa`.

A. Sampling

In the Sampling phase, `WPS` employs `StraSa` on the *WIT* to selectively process only one among several items of the same stratum at each level of the *WIT*. Following this technique, this phase produces a sampled tree ST and a set C of items that were encountered but not expanded. The subtree ST is used to estimate the size of subtree rooted at items of different strata (see Section V-B below), while the items in C are partitioned amongst the available processing nodes according to the size of the subtrees provided by ST (see Section V-C below).

`WPS` offers, through the stratification, a customizable labeling system to define properties that constitute two items to be of the same stratum. For instance, in an Iterative Deepening A* (IDA*) search tree, two items may be considered to belong to the same stratum if their h-values, *i.e.,* their estimated cost-to-goal, are the same.

B. Estimating

In this phase, `WPS` computes the estimated size of the subtrees rooted at each item $n \in ST$. To compute this estimate, `WPS` traverses the ST bottom up and uses dynamic programming, as shown in Algorithm 3. In Algorithm 3 the values of Y_u^i represent the estimated size of the subtree rooted at the node of stratum u at level i of the *WIT*. The traversal of the ST, represented by the structure A, starts at the deepest level and moves toward the root (line 2). The values of Y_u^{i+1} are used to compute the values of Y_u^i (line 6). In this phase `WPS` produces a collection χ of Y_u^i values for every u and i encountered in the ST.

Algorithm 3: ComputeSubtreeSizes

Input: sampled tree A and stratification T
Output: a collection χ of the estimated subtree sizes Y_t^i for each level i and stratum t in A.

1: $\chi \leftarrow \{\}$
2: **for** $i \leftarrow$ tree depth to 1 **do**
3:   **for** each item n in $A[i]$ **do**
4:     $Y_{T(n)}^i \leftarrow 1$
5:     **for** each child n'' of n in the *WIT* **do**
6:       $Y_{T(n)}^i \leftarrow Y_{T(n)}^i + Y_{T(n'')}^{i+1}$
7:     **end for**
8:     insert $Y_{T(n)}^i$ in χ
9:   **end for**
10:   $i \leftarrow i - 1$
11: **end for**

C. Partitioning

In the Sampling phase, `WPS` processes a small subset of the items in the *WIT* through `StraSa`. In this phase `WPS` partitions the remaining items in the *WIT* — the items not processed by `StraSa` — into M groups, where M is the number of processing nodes available. The items not processed by `StraSa` are the items in C as well as the items reachable from the items in C.

`StraSa` ensures that for each item n' in $C[i]$ there is a unique item n in $A[i]$ with $T(n) = T(n')$. Moreover, given Chen's assumption that items of the same stratum root subtrees of the same size, $Y_{T(n)}^i$ in χ is an estimate of the number of items in the subtree rooted at n' (number of items reachable from n'). Thus, at this point, the problem of evenly partitioning the workload reduces to the NP-Hard multi-way number partitioning problem [7]: the algorithm must partition the items n' in C into M parts $W_1, W_2, \cdots, W_M$ such that the sum of the $Y_{T(n')}^i$ values in each part W_j and W_k with $j, k \in \{1, 2, \cdots, M\}$ are as similar as possible to each other. `WPS` employs the Balanced Largest-First Differencing Method (BLDM) [16] to compute an approximated solution to the number partitioning problem. BLDM is a widely used and effective algorithm that performs k-way partitioning for $k \geq 2$ in $\mathcal{O}(n \log n)$ time.

This work uses the number of items rooted at each given item as a workload metric for even distribution. We assume that the time required to process an item is constant throughout the *WIT* — an assumption that holds in all the applications studied in this paper. `WPS` could be easily adapted to use other workload metrics as well. For example, in applications where work items have different processing times, `StraSa` could estimate the total processing time of subtrees as opposed to estimating the size of the subtrees. Then, BLDM would be used to partition the items not processed by `StraSa` into parts of similar processing time.

D. Distributing

In this phase, `WPS` stores one subset W_1 in local memory for processing in the current processing node and distributes the remaining $W_{j|j=2..M}$ subsets of items to the $M-1$ remaining processing nodes. The items are copied to the nodes asynchronously to ensure that a processing node does not need to wait for completion of data transfer to any other nodes. Multiple independent threads can be used to parallelize the copying operations across the processing nodes. In the applications used in this study, the work-items in different W_j subsets can be processed in any order as the applications generate valid results for all orders of processing of the items.

In applications, where the work-items must be expanded in an orderly fashion, processing of parent items first may be necessary to ensure that its children do not wait for a prolonged time. Work-items in such applications can be processed in a monotonically increasing order of $A[i]$ because the parent items are stored at higher levels of the ST than their children. Optimizing the scheduling and distribution strategy for applications that exhibit irregular dependencies among the work items is beyond the scope of this paper.

VI. WPS in the Multiple-Probing Setting

Typically, increasing the number of `StraSa` probes will improve the accuracy of the tree size prediction. In `WPS` we use multiple probes to improve the prediction accuracy of the size of the subtrees rooted at items of different strata. In the partitioning phase, for stratum t encountered at level i, instead of using the Y_t^i value produced in a single probe, we use the average of the Y_t^i-values computed across multiple probes.

An important question is: how many probes are sufficient to yield best performance? An empirical evaluation indicates that a larger number of probes improves the accuracy of estimation, but incurs large sampling overhead. Likewise, a smaller number of probes with lower overhead may lead to poor estimates of the workload metric. This study used manual tuning to determine the number of probes that yields the best performance (details in Section XI). The findings from this study establish the significant performance merit of the sampling-based workload distribution technique. Future studies may use automatic techniques to identify the optimal number of probes for high performance. For instance, they could use profile-guided tuning, where `WPS` could continue the probing process until the execution time stops improving.

VII. WPS Accuracy

`WPS` is intended to evenly distribute the workload among different processing nodes in a cluster, thus minimizing the need for load-balancing operations. An empirical evaluation (Section XI) indicates that `WPS` performs a good partitioning of the work list and consequently requires infrequent load-balancing operations. However, the approximations of the stratified sampling technique and of the multi-way number partitioning algorithm may leave room for some load imbalance both inside and across multiple nodes in a cluster. Therefore, `WPS` is intended to operate in coordination with existing load-balancing schedulers, such as work stealing, that manage intra- and inter-node load imbalances, and not to replace them completely. For the applications studied in

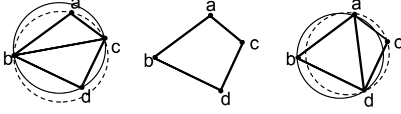

Fig. 2: Triangle-flipping process.

this implementation, there was limited scope for inter-node load balancing. Nonetheless, coordinating WPS with existing load-balancing schemes in runtimes of programming systems is beneficial because: i) there is no load-balancing overhead if there is no load imbalance in the system; and ii) the infrequent load-balancing operations that may be necessary will be handled by the existing load-balancing techniques.

VIII. Application Problems

We apply WPS to parallelize three algorithms: IDA*, Delaunay Mesh Generation, and Delaunay Mesh Refinement.

A. Iterative-Deepening A* (IDA*)

IDA* is a fundamental algorithm in Artificial Intelligence for solving state-space search problems. Given a start state s^*, IDA* expands a tree while performing a Depth-First Search from s^* with cost bound d in the state space. IDA* uses a cost function defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach state n from s^*, and $h(n)$ is the estimated cost-to-go from n. The value of the cost bound d is initially set to the heuristic value of s^*.

In each iteration, IDA* expands all states n that it encounters such that $f(n) \leq d$. If a goal is not found, then d is increased by setting it to the lowest f-value larger than d observed in the previous iteration. If IDA* uses an admissible heuristic — a heuristic that never overestimates the optimal solution cost for any state n — then IDA* is guaranteed to return a path from s^* to the goal state, if one exists, with the optimal solution cost [12].

1) Parallelizing IDA:* The tree that IDA* expands during its search, with a given cost bound, is WPS's ***WIT***. The states in this tree represent the items. In each IDA* iteration, WPS partitions the ***WIT*** into M parts of similar size. The M processors detect termination of each iteration and compute an estimate of the cost for the next iteration. WPS is used once again to partition the new ***WIT*** defined by the new cost bound. If a processor finishes its part of the search, it tries to steal items from other processors. All processors stop once the solution is found. Processor idling leads to substantial performance degradation because an iteration of IDA* does not start until the previous one is completed. Therefore, a balanced workload partitioning scheme is crucial for agent-search domains such as IDA*.

B. Delaunay Mesh Generation (DMG)

Delaunay Triangulation, *a.k.a Delaunay Mesh Generation* (DMG), from a set of points P in an Euclidean plane is a triangulation DT such that no point in P is inside the circumcircle of any triangle in DT. DMG starts by adding to P three dummy points *at infinity*, whose spanning triangle contains all points in P. In each iteration, the algorithm inserts a point $p \in P$ into the current triangulation by connecting p to the three vertices of the triangulation. The new triangles formed after triangulation are valid Delaunay triangles iff there is no point inside the circumcircle of any of the triangles.

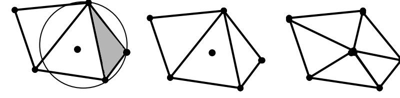
Fig. 3: Retriangulating a bad triangle in DMR.

The first triangulation in Figure 2 is an example of an invalid triangulation because the circumcircle of Δ_{bcd} encompasses the point a, which is a vertex of the other Δ_{abc}. Therefore, the triangulation process deletes the edge bc, shared by Δ_{abc} and Δ_{bcd} (second diagram in the figure), connects a to the vertex d of Δ_{bcd} (third triangulation in the figure), and examines the newly formed Δ_{acd} and Δ_{abd} for validity. This process, called *triangle-flipping*, continues until all triangles encountered during triangulation are valid. The triangulation obtained after insertion of all points is the Delaunay mesh. The final triangulation shown in Figure 2 is valid because there is no point inside the circumcircle of any of the triangles.

1) Parallelizing DMG: Processing of points during Delaunay Triangulation can be done in an arbitray order — all orders generate valid Delaunay meshes. Processing a point involves splitting a triangle, and possibly, flipping invalid triangles in the point's neighbourhood. Typically, these neighbourhoods are small – the connected regions of the mesh. DMG can be parallelized by inserting multiple points in parallel, provided the points affect triangles that are far apart in the mesh.

The items to be balanced in DMG are the set of given points. The child triangles resulting from the insertion of different points into a given triangle form WPS's ***WIT***. WPS first generates sufficient triangles in ***WIT*** for M processing nodes and partitions the ***WIT*** into M parts of similar size. A processor tries to steal items from other processors if it completes triangulations of its allocated ***WIT***. All processors stop once all given points have been processed.

C. Delaunay Mesh Refinement (DMR)

Delaunay Mesh Refinement (DMR) refines the given Delaunay Triangulation such that no angle in any of the triangles in the mesh is less than 30 degrees. The triangles that do not meet this criterion are *bad* triangles. In each iteration, DMR successively fixes the bad triangles by adding new points to the mesh and re-triangulating the resulting triangles. Figure 3 illustrates this process. For example, the shaded triangle in Figure 3 is a bad triangle. To fix this bad triangle, a new point is added at the circumcenter of this triangle. Adding this point may invalidate the Delaunay property of some triangles in the neighbourhood of this triangle. This region is called the *cavity* of the bad triangle. Re-triangulating a cavity to fix the bad triangles may generate new bad triangles, but this iterative refinement process ultimately terminates and produces a guaranteed-quality mesh.

1) Parallelizing DMR: A *cavity* formed during the processing of a bad triangle is typically a small neighbourhood of the

TABLE I: Instances of 24-puzzles solved optimally.

13 14 17 22 9 21 8 10 6 7 5 16 0 24 1 15 2 23 4 3 18 19 12 11 20
2 0 10 19 1 4 16 3 15 20 22 9 6 18 5 13 12 21 8 17 23 11 24 7 14
9 6 15 10 0 20 17 16 5 24 2 3 21 14 7 18 13 19 4 12 11 22 8 23 1

triangle. The small size of cavities implies that two bad triangles that are far apart on the mesh may have non-overlapping cavities. Furthermore, the entire process of expanding a bad triangle, re-triangulating the cavity and updating the mesh are completely independent for two triangles. Thus, multiple triangles with non-overlapping cavities can be processed in parallel. However, two triangles with overlapping cavities must be processed sequentially. The correctness of the final refined mesh is agnostic to the order of processing of the triangles — all orders lead to valid triangulations.

The items to be balanced are the bad triangles in the given Delaunay Triangulation. The cavities formed while iteratively fixing different bad triangles form `WPS`'s WIT.

IX. The X10 Programming System

The applications and the runtime changes are implemented in X10 [26], a well-supported high-performance programming system. The X10 programming model is organized around the notions of *activities* and *places*. Every computation in X10 is an asynchronous activity, akin to a light-weight task. A place is an abstraction of shared, mutable data and worker threads operating on the data. It encodes the affinity between tasks and memory partitions. Every activity runs in a place. In X10 the statement `async (p) S` creates a new activity at place `p` to execute `S`. The activities running in a place may access co-located data with the efficiency of local access. An access to a remote place may take orders of magnitude longer and is performed using the `at (p) S` statement. The `at` statement shifts the control of execution of the current activity from the current place to place `p`, copies any data that is required by the statements `S` to `p`, and, at the end, returns the control of execution to the original place. The data copying is done through runtime system calls inserted by the compiler.

X10 uses an intra-node work-stealing scheduler for load balancing. The scheduler uses a pool of worker threads to execute activities. Each worker owns a deque — double-ended queue — of activities. A worker pushes an activity for each `async` construct it encounters. When a worker completes one activity, it pops the next activity to run from its deque. A worker attempts to steal an activity from the deque of another worker if its own deque is empty. Contention is minimal and only arises with load imbalance because each worker primarily interacts with its own deque. Contention is further reduced because the push and pop operations access one end of the deque while steal operations access the other end.

X. Experimental Setup

Platform Performance measurements use a blade server with 16 nodes, each featuring two 2 GHz Quad-Core AMD Opteron processors, with 8 GB of RAM and 20 GB of swap space, running CentOS GNU/Linux version 6.2. All binaries were compiled with GCC version 4.4.3 using the -O3 flag.

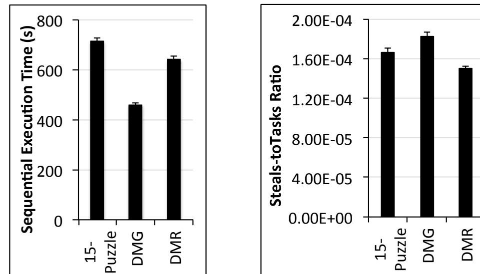

Fig. 4: Sequential execution times and steal-to-task ratios.

Compiler and Runtime The `x10c++` compiler version 2.3.1 is used for all measurements. The nodes in the cluster are connected by an infiniband network with a bandwidth of 10 Gbit/s and use MVAPICH2 library for communication. The experimental runs create eight worker threads per node and vary the number of nodes from 1 to 16 so that the number of threads is the same as the total number of cores.

Start States for WPS This evaluation uses Korf's [11] one hundred instances of 15-puzzle and 3 random instances of 24-puzzle (shown in Table I). The start states for the puzzles can be generated through random valid permutations of the tiles. There is only one valid start state for DMG — the triangulation that encapsulates the given points. For DMR, triangulation of any given bad triangle is a valid start state. DMG uses 10M points and DMR uses 68M triangles as inputs.

Stratification Our evaluation uses a stratification that labels items n_1 and n_2 with the same stratum if three conditions are met: (1) $h(n_1) = h(n_2)$; (2) n_1 and n_2 generate the same number of children and grandchildren; and (3) the children and grandchildren have the same heuristic values. Lelis *et. al* [15] first introduced this stratification and showed that `StraSa` produces good estimates of the IDA* *WIT* size using such a stratification. For DMG and DMR, two items are said to have the same stratum if they generate the same number of children and grandchildren. Unless stated otherwise, `WPS` employs five probes of `StraSa` on 15-Puzzle, DMG, and DMR, and 25 probes on 24-Puzzle. Section XI discusses the performance impacts of different stratifications and number of probes.

Methodology Applications are run twenty times to account for variances, such as work-stealing in the X10 runtime, and scheduling policies in the operating system. The performance charts include 95% confidence intervals for execution times, speedups, and steals-to-tasks ratios.

Application Properties The ratio of the total number of tasks stolen to the total number of tasks created during the execution of an application is its steal-to-task ratio. Figure 4 shows the steal-to-task ratio for the applications at 128 threads using `X10WS`. Although the ratios are small, the absolute numbers of work-stealing operations required for load balancing are high — 32,093 for 15-puzzle, 776,193 for one instance of the 24-puzzle, 63,290 for DMG and 52,647 for DMR — indicating that these applications can substantially benefit from a more balanced workload distribution.

Workload Distribution Algorithms The idea of performing state-space exploration to generate enough parallel tasks for distribution amongst processing nodes is not new. `WPS` pro-

vides a systematic way to do this without requiring programmers to code this solution. Unlike other existing techniques, WPS estimates the number of tasks that will be generated after processing the initial tasks to evenly divide the total workload in an application. This evaluation compares the performance of WPS against the following workload-distribution algorithms:

i) X10WS: X10's default intra-node work-stealing scheduler
ii) WD scheduler complements X10WS and migrates tasks to a node whose worker repeatedly fails to steal from its co-located peers [19], [18].
iii) EagerWD scheduler is similar to WD but proactively maps tasks in a load-balance-aware manner rather than waiting until the occurrence of a load-imbalance.
iv) DistWS scheduler complements X10's intra-node work-stealing with inter-node work-stealing [20].

The evaluation also investigates the implications of operating WPS in isolation and in tandem with these schedulers:

v) WPS*: WPS operating without coordination with any other scheduler;
vi) WPS: WPS operating in coordination with X10WS; and
vii) WPS+DistWS: WPS operating in coordination with both X10WS and DistWS.

A closely-related algorithm intended specifically for IDA* also exists, it is called AIDA*. Section XII compares the performance of WPS against AIDA*.

Initial Partitioning of Work-List WPS partitions the *WIT* based on the predicted size of the subtrees rooted at items in the *WIT*. X10WS and DistWS employ the following approach to partitioning. The sliding-tile and DMG applications start with a single state and generate several states during the course of their execution. Therefore, in 15- and 24-Puzzles, the algorithms perform an iterative-deepening search for a few levels until there are at least $10 \times M$ items in the search frontier, where M is the number of available processing nodes. This initial exploration produces a sufficient number of subtree roots — about 1,300 nodes — while not overflowing the memory resources of the evaluation platform. Similarly, in DMG, the algorithms perform triangulation until $10 \times M$ triangles are formed in the mesh. These triangles and their encapsulated points are then distributed among M processing nodes. In DMR, the implicitly-defined worklist of triangles to be refined is distributed amongst M processing nodes. Unlike WPS, X10WS and DistWS are unable to account for the dynamically generated items during workload distribution.

XI. Evaluation

The sequential execution times of the three instances of 24-puzzle are long and they significantly differ from one another. Therefore, the parallel execution time for each instance at 128 workers is reported separately in Figure 5. The speedup results for other applications are shown in Figure 6. This evaluation investigates the following research questions:

(1) How beneficial is WPS*? For each application, Figure 6 shows the speedups achieved using different workload-distribution strategies at different worker counts. The speedups are relative to the sequential execution time using X10WS.

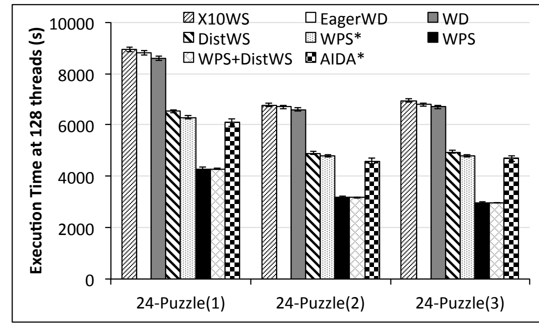

Fig. 5: Execution times of 24-Puzzle at 128 workers.

WPS* *consistently outperforms* EagerWD, WD, *and* DistWS. For instance, the 86x speedup on DMR at 128 workers achieved using WPS* represents 23%, 20%, and 18% improvements over EagerWD, WD, and DistWS, respectively. EagerWD and WD yield relatively small speedups over X10WS. EagerWD yields 11%, 10%, and 14% speedups on 15-puzzle, DMG and DMR respectively. WD exhibits better speedups — 14%, 16%, and 19% on the same applications. DistWS performs much better. It yields speedups of 22%, 32% and 29%. The algorithms exhibit similar trend on 24-Puzzles too. DistWS outperforms X10WS by 26%, 27%, and 28% on three instances of 24-Puzzle. WPS* outperforms X10WS by 29%, 29%, and 31% on the same instances.

(2) What is the impact of coordinating WPS* with other schedulers? WPS* aims to evenly distribute the workload among processing nodes, but not among multiple threads within a processing node. Therefore, WPS scheduler coordinates WPS* with X10WS. WPS yields significant speedups over X10WS on 15-Puzzle, DMG, and DMR — 38%, 45%, and 43% respectively. WPS *outperforms* DistWS*— the best performing scheduler among* X10WS, EagerWD, WD, *and* DistWS*— by 21%, 18%, and 20% on 15-Puzzle, DMG, and DMR respectively.* WPS outperforms DistWS by 34%, 35%, and 40% on three instances of 24-Puzzle.

WPS+DistWS does not yield significant speedup over WPS. This is a result of the even workload distribution generated by WPS, which leaves little opportunities for DistWS to migrate items between nodes for load balancing.

Further evaluations do not discuss WPS* and WPS+DistWS because they are not interesting in terms of overall speedups.

(3) How does the precision of stratification impact WPS? The quality of the stratification guiding the sampling process may substantially impact the accuracy of the tree-size predictions. Ideally, a stratification would determine that two items are of the same stratum iff they root subtrees of the same size, which would allow StraSa to produce a perfect estimate of the tree size in a single probe. A trivial example of such a stratification is one in which every item belongs to its own stratum. In this case, there would be far too many strata to sample from and the approach would not be effective. In practice one should use a compact stratification (i.e., a stratification with a small number of strata) that has a low variance on the subtree sizes rooted at items of the same stratum. Multiple StraSa probes can be used to improve the

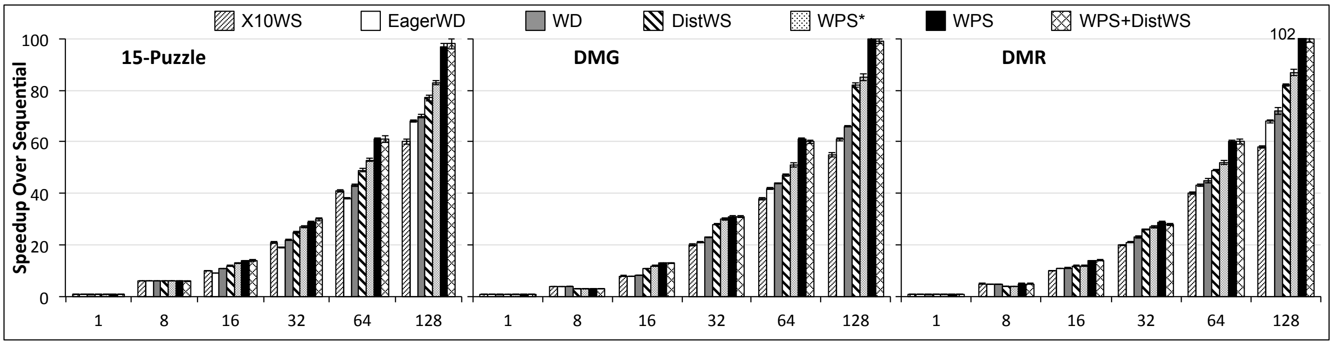

Fig. 6: Application speedups over sequential execution times using different schedulers at different worker counts.

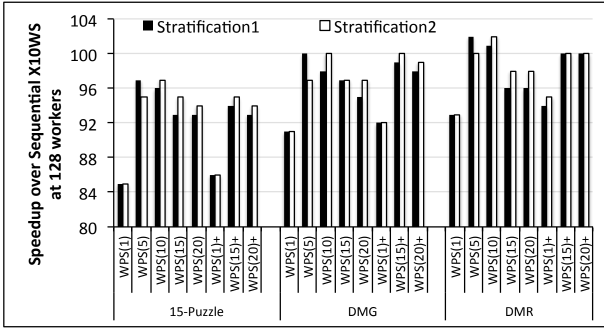

Fig. 7: WPS performance using different stratifications. The number of probes used are indicated in the parentheses, and the '+' sign in the x-axis labels indicate WPS+DistWS.

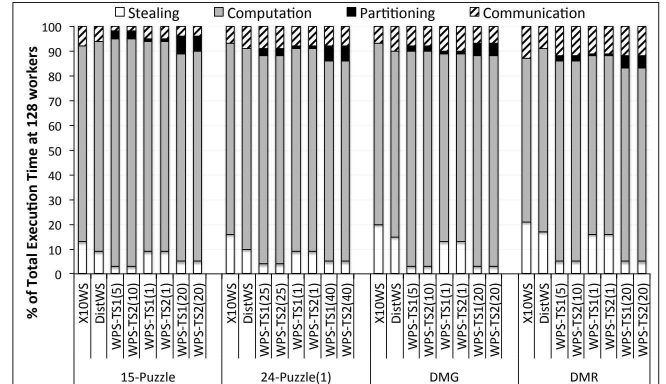

Fig. 8: Breakdown of total execution time.

accuracy of predictions when using low-quality stratifications.

Figure 7 shows the performance impact of different stratifications and of different number of StraSa probes on WPS. Stratification1 (S1) is the stratification described in Section X, while Stratification2 (S2) is a version of S1 with fewer strata. Namely, S2 considers two items to be of different stratum if the number of children and grandchildren they produce differ by more than two. The numbers in parentheses in Figure 7 represent the number of probes used. WPS performs worse in all domains when using one probe (see WPS(1)). When using a single probe, the performance does not improve even when coordinating WPS with DistWS (see WPS(1)+).

WPS yields best performance under S1 and S2 at five and ten probes of StraSa, respectively. This difference in the number of probes in which each stratification performs best is because S1 has more strata than S2. Thus, a StraSa probe using S2 will tend to be faster than a StraSa probe using S1. Increasing the number of probes beyond five for S1 and ten for S2 is not beneficial. In fact, the performance worsened at larger number of probes because of the overhead of sampling. The loss in performance is regained by coordinating WPS with DistWS. The 24-Puzzles exhibit best performance under both stratifications at 25 probes, and exhibit similar trends overall.

(4) What are the sources of speedups? Fig. 8 shows the breakdown of execution times of applications using X10WS, DistWS, and WPS. The performance gains arise from:

i) *Reduced Work-stealing Operations:* WPS reduces the execution time spent on work stealing in the range of 10% to 16% over X10WS and in the range of 7% to 12% over DistWS. The work-stealing time also accounts for the machine idle times — when workers are unsuccessfully searching for surplus work to steal — because an idle worker may continuously try to steal work from other workers. The reduced need for work-stealing operations means that the nodes will be mostly performing useful computations. Thus, actual computations in applications contribute more to the total execution time with WPS – in the range of 7% to 13% over DistWS and in the range of 14% to 16% over X10WS. These performance gains are achieved at the cost of 1% to 2% of the total execution time spent on workload partitioning using WPS.

ii) *Reduced Communication over the Network:* An even distribution of workload using WPS necessitates fewer message transmissions across the network. This reduced communication stems from fewer steal operations, fewer synchronized access to the shared deques of remote workers, and fewer accesses to data required to process stolen tasks.

Table II shows the average number of messages transmitted across the network obtained from twenty runs of each application using X10WS, DistWS, and WPS at 128 threads. As expected, WPS requires fewer message transmissions across the network compared to X10WS and DistWS.

iii) *Improved Node Utilization:* Through an improved workload distribution, significantly reduced work-stealing operations, and reduced machine idle times, WPS achieves an increased and uniform CPU utilization compared to X10WS and DistWS. The radial axes and the points in the circumference

TABLE II: Messages transmitted across network (in millions).

Applications	# of Messages Transmitted (128 workers)		
	X10WS	DistWS	WPS
15-Puzzle	12.43	10.49	3.40
24-Puzzle(1)	86.39	81.78	69.58
DMG	42.68	36.84	26.03
DMR	37.92	32.29	22.89

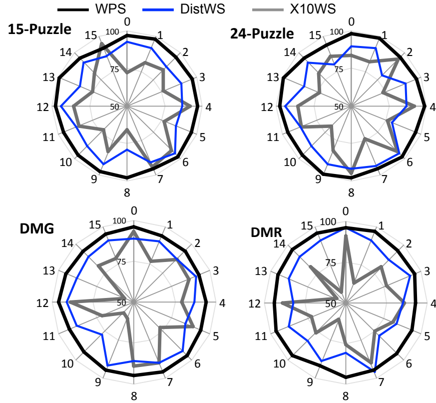

Fig. 9: Average node utilization (128 threads). The figure shows the node utilizations starting from 50% for better clarity.

of the circles in Fig. 9 respectively indicate the average CPU utilization of eight cores in a node and a node in the cluster.

With X10WS, the standard deviations of average node utilization for 15-Puzzle, DMG and DMR are 12.45, 18.37, and 17.5 respectively. With DistWS, the standard deviations are 9.71, 11.49, and 10.2 respectively. With WPS, the standard deviations are 2.32, 3.12, 3.09 respectively. The lower standard deviations of average node-utilizations when using WPS point to an improved load-distribution achieved with WPS.

DistWS permits workers in a node to steal tasks from remote nodes if all the co-located workers lack surplus work. Thus, DistWS performs more work compared to X10WS, which operates only within a node, resulting in an increased CPU utilization over X10WS. However, DistWS still exhibits a non-uniform CPU utilization due to a large number of tasks stolen for load-balancing. Stealing tasks from remote nodes incurs the overheads of remote data accesses that are necessary for processing the stolen tasks. Hence, some nodes show heavy utilization while the others show lighter utilization. WPS reduces the need for work stealing across processing nodes, thereby, enabling workers to perform useful computations. Thus, the node utilization circles for WPS are larger and more uniform compared to those for X10WS and DistWS.

XII. Asynchronous IDA* (AIDA*)

Similar to WPS, AIDA* also combines a data-partitioning scheme with work stealing for parallel and distributed implementation of IDA* [24]. AIDA* operates in three phases:

i) in the *data-partitioning phase*, each processor redundantly expands the tree to generate enough frontier nodes;
ii) in the *distributed node-expansion phase*, each processor expands its portion of the nodes generated in the preceding phase to generate additional finer-grained nodes;

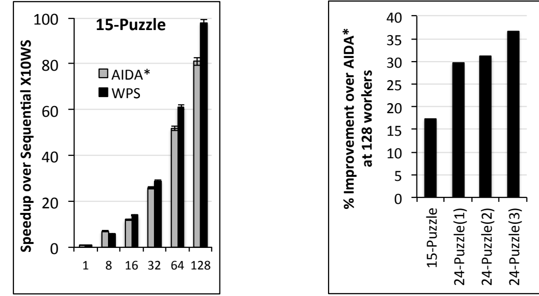

Fig. 10: WPS and AIDA* performance on 15- and 24-Puzzles.

iii) in the *asynchronous search phase*, processors perform IDA* on their subtrees until a solution is found. All processors search to the same threshold. After completion of an iteration, the processors begin a new search pass through the same set of subtrees using a larger threshold.

This implementation of AIDA* uses the following constraints to closely match the original algorithm: i) steals from neighbouring processors; ii) partially re-orders the nodes in the local work list and only allows nodes of average size to be stolen; iii) steals in chunk sizes of five nodes; iv) permits stealing of at most half of a victim's items. The cluster used in this evaluation is fully connected, where stealing from a randomly selected victim has the same effect as trying to steal from a neighbour in terms of communication latency. Nevertheless, we modify our load-balancer to mimic the original algorithm.

AIDA* differs from WPS as follows: i) unlike AIDA*, WPS expands the search tree on a single processor to generate enough frontier items for distribution among processors; ii) while AIDA* relies on a robust distributed load balancer to mitigate the imbalance in subtrees distributed to the processors, WPS rarely needs load balancing across processing nodes. WPS relies only on intra-node load balancer to mitigate load imbalance among workers co-located in a processing node.

For the three instances of 24-Puzzle, Figure 5 shows the execution time performance using AIDA* and WPS. For the 15-Puzzle, Figure 10 shows the speedups achieved with AIDA* and WPS at different worker counts. Figure 10 also shows the performance gains with WPS relative to AIDA* at 128 workers. AIDA* does not account for the variability in the subtree sizes during initial distribution of the frontier nodes. Consequently, processing nodes often suffer from frequent load imbalances and require several expensive load-balancing operations. In addition, AIDA* also needs to perform expensive item sorting and coalescing to support efficient migration of items. As a result, WPS outperforms AIDA* by 17%, 29%, 31% and 36% on 15-Puzzle, and the three instances of 24-Puzzle.

XIII. Additional Related Work

A popular task-distribution strategy in agent-search domains expands the search tree to generate enough frontier nodes for distribution among processors [23], [14]. Such a strategy does not account for the size of dynamically generated states, thereby, causing a load imbalance. Parallel Window Search is another approach for load balancing where multiple worker

threads search the same tree in parallel using different cost bounds [22]. The demerit of this approach is that some processors may search the tree with a search bound that is too high and result in wasted work if the optimal solution is found at lower depths by some other workers.

Transposition-table-driven scheduling (TDS) uses a hash-function-indexed transposition table for workload distribution [25]. TDS relies on robust optimizations, such as message coalescing, and requires: (i) dynamically identifying states with identical source and destination processors; (ii) coalescing them into a single message of appropriate granularity; and (iii) migrating the coalesced message packets at precise control-flow points. Efficient implementation of such operations often requires expensive data and control flow analyses.

Niewiadomski *et al.* [17] present a sampling-based approach for workload distribution in implicit graphs. Unlike their approach, `WPS` selectively samples only items of unique stratum and therefore does not require application programmers to manually identify the best sampling size for each application.

Frameworks, such as PREMA [1], Scioto [5], and Turbine [28], also aim to support automatic load balancing in irregular parallel applications. These frameworks require programmers to expose parallelism, and migratable data and objects in applications using special runtime libraries. Unlike these approaches, `WPS` does not require re-writing existing X10 applications, and only requires users to specify stratifications during program execution.

XIV. Concluding Remarks

This paper presents an algorithm for evenly partitioning workload in distributed shared-memory machines. The distribution relies on a sampling-based prediction of the size of the sub-tree rooted at a given item. An experimental evaluation of this approach on IDA*, Delaunay Triangulation and Delaunay Mesh Refinement algorithms yields substantial speedups compared to state-of-the-art approaches, including ones that are specifically targeted at individual applications.

The main strength of the algorithm is that its workload distribution strategy applies to a range of iterative work-list-based data-parallel irregular applications and does not require manual tuning of sampling strides. Another strength of the algorithm is its ability to integrate well with mainstream load-balancers, such as work-stealing schedulers. Although not necessary for applications studied in our evaluation, the algorithm may benefit from coordination with work-stealing schedulers, but it does not require a complete overhaul or removal of the existing schedulers. As such, this approach has wide scope in other high-performance programming systems such as Chapel [3], UPC [6], and Charm++ [9].

References

[1] K. Barker, A. N. Chernikov, N. Chrisochoides, and K. Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. *IEEE Trans. Parallel Distrib. Syst.*, 15:2:183–192, Feb. 2004.

[2] A. Bowyer. Computing Dirichlet Tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[3] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, Aug. 2007.

[4] P.-C. Chen. Heuristic Sampling: A Method for Predicting the Performance of Tree Searching Programs. *SIAM Journal on Computing*, 21:295–315, 1992.

[5] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A Framework for Global-View Task Parallelism. In *International Conference on Parallel Processing*, pages 586–593, Washington, DC, USA, 2008.

[6] T. El-Ghazawi and L. Smith. Upc: Unified parallel c. In *Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, San Francisco, 1979.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[9] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System based on C++. In *Conference on Object-oriented programming systems, languages, and applications*, pages 91–108, Washington, D.C., USA, 1993.

[10] D. E. Knuth. Estimating the Efficiency of Backtrack Programs. *Math. Comp.*, 29:121–136, 1975.

[11] R. E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.

[12] R. E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.

[13] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *Conference on Programming Language Design and Implementation*, pages 211–222, San Diego, California, USA, 2007. ACM.

[14] V. Kumar and V. N. Rao. Parallel Algorithms for Machine Intelligence and Vision. chapter Scalable Parallel Formulations of Depth-first Search, pages 1–41. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[15] L. H. S. Lelis, S. Zilles, and R. C. Holte. Predicting the Size of IDA*'s Search Tree. *Artificial Intelligence*, pages 53–76, 2013.

[16] W. Michiels, J. Korst, E. Aarts, and J. Leeuwen. Performance Ratios for the Differencing Method Applied to the Balanced Number Partitioning Problem. In *STACS 2003*, volume 2607 of *Lecture Notes in Computer Science*, pages 583–595. Springer Berlin Heidelberg, 2003.

[17] R. Niewiadomski, J. N. Amaral, and R. Holte. A Parallel External-Memory Frontier Breadth-First Traversal Algorithm for Clusters of Workstations. In *Int. Conf. on Parallel Pro.*, pages 531–538, Aug 2006.

[18] J. Paudel and J. N. Amaral. Hybrid Parallel Task Placement in Irregular Applications. *Journal of Parallel and Distributed Computing*, 2014. doi:10.1016/j.jpdc.2014.09.014.

[19] J. Paudel, O. Tardieu, and J. N. Amaral. Hybrid Parallel Task Placement in X10. In *X10 Workshop*, pages 31–38, Seattle, Washington, 2013.

[20] J. Paudel, O. Tardieu, and J. N. Amaral. On the Merits of Distributed Work-Stealing on Selective Locality-Aware Tasks. In *International Conf. on Parallel Processing*, pages 100–109, Lyon, France, 2013.

[21] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison & Wesley, 1984.

[22] C. Powley and R. Korf. Single-Agent Parallel Window Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):466–477, May 1991.

[23] V. N. Rao, V. Kumar, and K. Ramesh. A Parallel Implementation of Iterative-Deepening-A. In *National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 178–182. AAAI Press, 1987.

[24] A. Reinefeld and V. Schnecke. AIDA* – Asynchronous Parallel IDA*. In *Canadian Conference on Artificial Intelligence*, pages 295–302, 1994.

[25] J. Romein, H. Bal, J. Schaeffer, and A. Plaat. A Performance Analysis of Transposition-Table-Driven Work Scheduling in Distributed Search. *IEEE Transact. on Parallel and Dist. Sys.*, 13(5):447–459, May 2002.

[26] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification. http://x10.codehaus.org/x10/documentation.

[27] D. F. Watson. Computing the n-Dimensional Delaunay Tessellation with Application to Voronoi Polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[28] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster. Turbine: A Distributed-memory Dataflow Engine for Extreme-scale Many-task Applications. In *Workshop on Scalable Workflow Execution Engines and Technologies*, pages 5:1–5:12, Scottsdale, Arizona, 2012.