Confidence-based Work Stealing in Parallel Constraint Programming

Geoffrey Chu[1], Christian Schulte[2], and Peter J. Stuckey[1]

[1] National ICT Australia, Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au
[2] KTH – Royal Institute of Technology, Sweden
cschulte@kth.se

Abstract. The most popular architecture for parallel search is work stealing: threads that have run out of work (nodes to be searched) steal from threads that still have work. Work stealing not only allows for dynamic load balancing, but also determines which parts of the search tree are searched next. Thus the place from where work is stolen has a dramatic effect on the efficiency of a parallel search algorithm.
This paper examines quantitatively how optimal work stealing can be performed given an estimate of the relative solution densities of the subtrees at each search tree node and relates it to the branching heuristic strength. An adaptive work stealing algorithm is presented that automatically performs different work stealing strategies based on the confidence of the branching heuristic at each node. Many parallel depth-first search patterns arise naturally from this algorithm. The algorithm produces near perfect or super linear algorithmic efficiencies on all problems tested. Real speedups using 8 threads range from 7 times to super linear.

1 Introduction

Architectures for parallel search in constraint programming typically use work stealing for distributing work (nodes to be searched) from running to idle threads, see for example [1–4]. Work stealing has often focused on keeping processors occupied. Its analysis often assumes that the amount of work to be done is fixed and independent of the work stealing scheme, for example [5].

While this is true for certain kinds of problems (finding all solutions, proving unsatisfiability), it is not true for others (finding a first solution, or finding an optimal solution). Such analyses fail to account for the fact that the place in the search tree from which work is stolen determines the search strategy and hence is bound to have a dramatic effect on efficiency.

Many approaches choose to steal from as close to the root of the search tree as possible, e.g. [3], as this tends to give the greatest granularity of work and minimizes the overhead of work stealing. However, this is not always the best strategy in terms of efficiency.

Effect of work stealing. Let us consider a relatively simple framework for parallel search. One thread begins with ownership of the entire search tree. When a thread finishes searching the subtree it was responsible for, it will steal an unexplored part of the search tree (a *job*) from its current owner. This continues until a solution is found or the entire search tree has been searched.

Example 1. The first problem we will consider is a simple model for small instances of the Traveling Salesman Problem, similar to [6]. In our experiments, with 8 threads, stealing left and low (as deep in the tree as possible) requires visiting a number of nodes equal to the sequential algorithm, while stealing high (near the root) requires visiting $\sim 30\%$ more nodes on average (see Table 1).

The explanation is simple. In one example instance, the sequential algorithm finds the optimal solution after 47 seconds of CPU time, after which it spends another ~ 300 seconds proving that no better solution exists. When the parallel algorithm is stealing left and low, all threads work towards finding that leftmost optimal solution, and the optimal solution is found in 47 seconds of CPU time as before (wall clock time $47/8 \approx 6$ seconds). After this, the search takes another 300 seconds of CPU time to conclude. Thus we have perfect linear speedup both in finding the optimal solution, and in proving that no better solution exist.

However, when the parallel algorithm is stealing high, only one thread actually explores the leftmost part of the search tree and works towards that leftmost optimal solution. The other seven threads explore other parts of the search tree, unfruitfully in this case. This time, the optimal solution is found in 47 seconds of *wall clock* time (376 seconds of CPU time!). The algorithm then spends another 200 seconds of CPU time proving optimality. That is, there is no speedup whatsoever for finding the optimal solution, but linear speedup for proving optimality. Since the optimal solution has been found so much later (376 seconds CPU time instead of 47 seconds), the threads search without the pruning benefits of the optimal solution. Hence, the number of nodes searched drastically increases, leading to a great loss of efficiency. Clearly, this effect gets worse as the number of threads increases. □

It may appear from this example that stealing left and low would be efficient for all problems. However, such a strategy can produce at best linear speedup.

Example 2. Consider the n-Queens problem. The search tree is very deep and a top level mistake in the branching will not be recovered from for hours.

Stealing low solves an instance within the time limit iff sequential depth first search solves it within the time limit. This is the case when a solution is in the very leftmost part of the search tree (only 4 instances out of 100, see Table 2).

Stealing high, in contrast, allows many areas of the search tree to be explored, so a poor choice at the root of the search tree is not as important. Stealing high results in solving 100 out of 100 instances tested. This is clearly far more robust than stealing low, producing greatly super-linear speedup. □

Veron *et al* [7] claim that linear and super linear speedups can be expected for branch and bound problems, but they fail to note that finding the optimal solution does not parallelize trivially as shown by Example 1. Rao and Kumar [8] (and others) show that super linear speedup ought to be consistently attainable for finding the first or the optimal solution for certain types of problems. Their analysis is valid if the search tree is random (i.e. the solutions are randomly distributed), but is not valid in systems where a branching heuristic orders the branches based on their likelihood of yielding a solution. The presence of such a branching heuristic makes linear speedup in finding solutions non-trivial. Gendron and Crainic [9] describe the issue and provide a description how the issue is handled in several systems. In general, the solutions utilise some kind of best-first criterion to guide how the problem is split up (see e.g. [10, 11]).

Contributions. The paper contributes a quantitative analysis of how different work stealing strategies affect the total amount of work performed and explains the relationship between branching heuristic strength and optimal search strategy. It introduces *confidence-based work stealing* as an adaptive algorithm that, when provided with a user-defined *confidence*, will steal work in a near optimal manner. The confidence is the estimated ratio of solution densities between the subtrees at each node. The paper shows that confidence-based work stealing leads to very good algorithmic efficiencies, that is, not many more, and sometimes much less, nodes are explored than for sequential DFS (Depth First Search).

Although the analysis is done in the context of parallel search for constraint programming, the analysis is actually about the relationship between branching heuristic strength and the optimal search order created by that branching heuristic. Thus the analysis actually applies to all complete tree search algorithms whether sequential or parallel. The paper shows that when the assumptions about branching heuristic strength that lie behind standard sequential algorithms such as DFS, Interleaved Depth First Search (IDFS) [12], Limited Discrepancy Search (LDS) [13] or Depth-bounded Discrepancy Search (DDS) [14] are given to the algorithm as *confidence* estimates, the algorithm produces the exact same search patterns used in those algorithms. Thus the analysis and algorithm provides a framework which explains/unifies/produces all those standard search strategies. In contrast to the standard sequential algorithms which are based on rather simplistic assumptions about how branching heuristic strength varies in different parts of the search tree, our algorithm can adapt to branching heuristic strength on a node by node basis, potentially producing search patterns that are vastly superior to the standard ones. The algorithm is also fully parallel and thus the paper also presents parallel DFS, IDFS, LDS and DDS as well.

Confidence-based search shares the idea to explore more promising parts of the search tree first with other approaches such as impact-based search [15] and using solution counting [16] and constraint-level advice [17] for guiding search. However, there are two key differences. First, confidence-based search uses standard branchings (labelings) that define the shape of the search tree augmented by confidence. This makes the addition of confidence-based search to an existing constraint programming solver straightforward and allows us to reuse existing constraint models with user-defined branchings. Second, confidence-based search is designed to work in parallel.

2 Analysis of Work Allocation

In this section we show quantitatively that the strength of the branching heuristic determines the optimal place to steal work from. We will concentrate on the case of solving a satisfaction problem. The case for optimization is related since it is basically a series of satisfaction problems.

Preliminary definitions. A *constraint state* (C, D) consists of a system of constraints C over variables V with initial domain D assigning possible values $D(v)$ to each variable $v \in V$. The *propagation solver*, solv, repeatedly removes values from the domains of variables that cannot take part in the solution of some constraint $c \in C$, until no more values can be removed. It obtains a new domain $\text{solv}(C, D) = D'$. If D' assigns a variable the empty set the resulting state is a *failure state*. If D' assigns each variable a single value ($|D(v)| = 1, v \in V$) the resulting state is a *solution state*. Failure and solution states are *final states*.

Finite domain propagation interleaves propagation solving with search. Given a current (non-final) state (C, D) where $D = \text{solv}(C, D)$ the search process chooses a *search disjunction* $\vee_{i=1}^{n} c_i$ of *decision constraints* c_i $(1 \leq i \leq n)$ which is a consequence of the current state $C \wedge D$. The child states of (C, D) are calculated as $(C \wedge c_i, \text{solv}(C \wedge c_i, D))$, $1 \leq i \leq n$. Given a root state (C, D), this defines a *search tree* of states, where each non-final state is an internal node with children defined by the search disjunction and final states are leaves.

The *solution density* of a search tree T with k nodes and l solution state nodes is l/k.

Optimal split for binary nodes. For simplicity, assume that visiting each node in the search tree has roughly equal cost. Assuming an oracle that provides accurate information on solution density, work stealing from nodes whose subtrees have the highest solution densities will be optimal.

In practice however, the solution density estimates will not be perfect:

1. Any estimate of the solution density of a subtree will have a very high error, with a substantial chance that the solution density is actually zero.
2. The real solution densities, and hence the errors in the estimate, are highly correlated between nearby subtrees, as they share decision constraints from higher up in the tree.
3. The solution density estimate of a subtree should decrease as nodes in that subtree are examined without finding a solution. This is caused by:
   (a) As the most fruitful parts of the subtree are searched, the average solution density of the remaining nodes decrease.
   (b) The correlation between solution densities between nearby subtrees mean that the more nodes have failed in that subtree, the more likely the remaining nodes are to fail as well.

We have to take these issues into account when utilizing solutions densities to determine where to steal work. Given the actual solution density probability distribution for the two branches, we can calculate the expected number of nodes searched to find a solution. We derive the expression for a simple case. Suppose the solution density probability distribution is uniform, that is, it has equal probability of being any value between 0 and S where S is the solution density estimate. Let A and B be the solution density estimates for the left and right branch respectively, and assume a proportion p and $(1-p)$ of the processing power is sent down the left and right branch respectively. Then the expected number of nodes to be searched is given by the hybrid function:

$$f(A,B,p) = \begin{cases} \frac{1}{pA}(2+\ln(\frac{pA}{(1-p)B})) & \text{for } pA > (1-p)B \\ \frac{1}{(1-p)B}(2+\ln(\frac{(1-p)B}{pA})) & \text{otherwise} \end{cases} \tag{1}$$

The shape of this function does not depend on the absolute values of A and B (which only serve to scale the function), but on their ratio, thus the shape is fixed for any fixed value of $r = A/(A+B)$.

The value of p which minimizes the function for given value of r is well approximated by the straight line $p = r$. In fact, the value of the f function at $p = r$ is no more than 2% higher than the true minimum for any r over the range of $0.1 \leq r < 0.9$. For simplicity, we will make this approximation from now on. This means that it is near optimal to divide the amount of processing power according to the ratio of the solution density estimate for the two branches. For example, if $r = 0.9$, which means that A is 9 times as high as B, then it is near optimal to send 0.9 of our processing power down the left branch and 0.1 of our processing power down the right.

Branching confidence. Define the *confidence* of a branching heuristic at each node as the ratio $r = A/(A+B)$. The branching heuristic can be considered *strong* when $r \rightarrow 1$, that is the solution density estimate of the left branch is far greater than for the right branch. In other words, the heuristic is really good at shaping the search tree so that solutions are near the left. In this case, our analysis shows that since r is close to 1, we should allocate almost all our processing power to the left branch every time. This is equivalent to stealing work for search as left and as low as possible. The branching heuristic is *weak* when $r \approx 0.5$, that is, the solution density estimate of the left branch and right branch are similar because the branching heuristic has no insight into where the solutions are. In this case, our analysis shows that since $r = 0.5$, the processing power should be distributed evenly between left and right branches at each node. This is equivalent to stealing work for search as high as possible.

3 Adaptive Work Stealing

Our analysis shows that the optimal work stealing strategy depends on the strength of the branching heuristic. Since we have a quantitative understanding of how optimal work stealing is related to branching heuristic strength, we can design a search algorithm that can automatically adapt and produce “optimal” search patterns when given some indication of the strength of the branching heuristic by the problem model. In this section, we flesh out the theory and discuss the implementation details of the algorithm.

3.1 Dynamically Updating Solution Density Estimates

Now we examine how solution density estimates should be updated during search as more information becomes available.

First we need to relate the solution density estimate of a subtree with root (C, D) with the solution density estimate of its child subtrees (the subtrees rooted at its child states $(C \wedge c_i, \text{solv}(C \wedge c_i, D))$). Consider an n-ary node. Let the subtree have solution density estimate S. Let the child subtree at the i-th branch have solution density estimate A_i and have size (number of nodes) k_i. If S and A_i are estimates of average solution density, then clearly: $S = \sum_{i=1}^{n} A_i k_i / \sum_{i=1}^{n} k_i$, i.e. the average solution density of the subtree is the weighted average of the solution densities of its child subtrees.

Uncorrelated subtrees. Assuming no correlation between the solution densities of subtrees, we have that if the first j child subtrees have been searched unsuccessfully, then the updated solution density estimate is $S = \sum_{i=j+1}^{n} A_i k_i / \sum_{i=j+1}^{n} k_i$. Assuming that k_i are all approximately equal, then the expression simplifies to:

$$S = \sum_{i=j+1}^{n} A_i/(n-j)$$

For example, suppose $A_1 = 0.3, A_2 = 0.2, A_3 = 0.1$, then initially, $S = (0.3 + 0.2 + 0.1)/3 = 0.2$. After branch 1 is searched, we have $S = (0.2 + 0.1)/2 = 0.15$, and after branch 2 is searched, we have $S = (0.1)/1 = 0.1$. This has the effect of reducing S as the branches with the highest values of A_i are searched, as the average of the remaining branches will decrease.

Correlated subtrees. Now we consider the case where there are correlations between the solution density estimates of the child subtrees. The correlation is likely since all of the nodes in a subtree share the constraint C of the parent state. Since the correlation is difficult to model we pick a simple generic model. Suppose the solution density estimates for each child subtree is given by $A_i = \rho A_i'$, where ρ represents the effect on the solution density due to the constraint added at the parent node, and A_i' represents the effect on the solution density due to constraints added within branch i. Then ρ is a common factor in the solution density estimates for each branch and represents the correlation between them. We have that:

$$S = \frac{\sum_{i=1}^{n} A_i k_i}{\sum_{i=1}^{n} k_i} = \rho \frac{\sum_{i=1}^{n} A_i' k_i}{\sum_{i=1}^{n} k_i}.$$

Suppose that when j out of n of the branches have been searched without finding a solution, the value of ρ is updated to $\rho \frac{n-j}{n}$. This models the idea that the more branches have failed, the more likely it is that the constraint C added at the parent node has already made solutions unlikely or impossible. Then, after j branches have been searched, we have: $S = \rho\frac{n-j}{n}\sum_{i=j+1}^{n} A'_i k_i / \sum_{i=j+1}^{n} k_i$. Assuming that all k_i are approximately equal again, the expression simplifies to: $S = \rho\frac{n-j}{n}\sum_{i=j+1}^{n} A'_i/(n-j) = \frac{\rho}{n}\sum_{i=j+1}^{n} A'_i = \sum_{i=j+1}^{n} A_i/n$. Equivalently, we can write it as:

$$S = \frac{1}{n}\sum_{i=1}^{n} A_i \text{ and } A_i = 0 \text{ for } 1 \leq i \leq j \tag{2}$$

where we update A_i to 0 when branch i fails. The formula can be recursively applied to update the solution density estimates of any node in the tree given a change in solution density estimate in one of its subtrees.

Using confidence. In all of our results, the actual values of the solution densities are not required. We can formulate everything using *confidence*, the ratio between the solution densities of the different branches at each node. In terms of confidence, when a subtree is searched and fails the confidence values should be updated as follows:

Let r_i be the confidence value of the node i levels above the root of the failed subtree and r'_i be the updated confidence value. Let $\bar{r}_i = r_i, \bar{r}'_i = r'_i$ if the failed subtree is in the left branch of the node i levels above the root of the failed subtree and $\bar{r}_i = 1 - r_i, \bar{r}'_i = 1 - r'_i$ otherwise. Then:

$$\bar{r}'_i = (\bar{r}_i - \prod_{k=1}^{i} \bar{r}_i)/(1 - \prod_{k=1}^{i} \bar{r}_i) \tag{3}$$

3.2 Confidence Model

Given a confidence at each node, we now know how to steal work "optimally" and how to update confidence values as search proceeds. But how do we get an initial confidence at each node. Ideally, the problem modeler, with expert knowledge about the problem and the branching heuristic can develop a solution density heuristic that gives a confidence value at each node. However, this may not always happen, perhaps due to a lack of time or expertise. We can simplify things by using general confidence models. For example, we could assume that the confidence takes on an equal value *conf* for all nodes. This is sufficient to model general ideas like: the heuristic is strong or the heuristic is weak. Or we could have a confidence model that assigns $r = 0.5$ to the top d levels and $r = 0.99$ for the rest. This can model ideas like the heuristic is weak for the first d levels, but very strong after that, much like the assumptions used in DDS [14].

3.3 Algorithm

Given that we have a confidence value at each node, our confidence-based search algorithm will work as follows. The number of threads down each branch of a node is updated as the search progresses. When search for a subtree is finished, the confidence values of all nodes above the finished subtree are updated as described by Equation (3) above.

When work stealing is required, we start at the root of the tree. At each node we use the number of threads down each branch, the confidence value, to determine which branch to take. Given the number of threads down each branch is currently a and b respectively then if $|(a+1)/(a+b+1)-r| \leq |a/(a+b+1)-r|$ we go left, otherwise right (i.e., which move would split the work closer to the confidence value). We continue this process until we find an unexplored node, at which point we steal the subtree with that unexplored node as root.

There is an exception to this. Although we may sometimes want to steal as low as possible, we cannot steal too low, as then the granularity would become too small and communication costs will dominate the runtime. Thus we dynamically determine a granularity bound under which threads are not allowed to steal, e.g. 15 levels above the average fail depth. If the work stealing algorithm guides the work stealing to the level of the granularity bound, then the last unexplored node above the granularity bound is stolen instead. The granularity bound is dynamically adjusted to maintain a minimum average subtree size so that work stealing does not occur more often than a certain threshold.

Since the confidence values are constantly updated, the optimal places to search next changes as search progresses. In order for our algorithm to adapt quickly, we do not require a thread to finish the entire subtree it stole before stealing again. Instead, after a given *restart* time has passed, the thread returns the unexplored parts of its subtree to a master coordinating work stealing and steals work again from the top. The work frontier is stored at the master and the work is resumed when work stealing guides a thread to the area again (similar to the idea used in interleaving DFS [12]).

Example with reasonably high confidence. Suppose we know that the branching heuristic is reasonably strong, but not perfect. We may use $conf = 0.8$.

Suppose we have 8 threads. Initially, all confidence values are 0.8. When the 8 threads attempt to steal work at the root, thread 1 will go down the left hand side. Thread 2 will go down the left hand side as well. Thread 3 will go down the right hand side. Thread 4 will go down the left hand side, etc, until 6 threads are down left and 2 threads are down right. At node b, we will have 5 threads down the left and 1 thread down the right, and so on. The work stealing has strongly favored sending threads towards the left side of each node because of the reasonably high confidence values of 0.8. The result is shown in Figure 1(a).

Suppose as search progresses the subtree starting at node d finishes without producing a solution. Then we need to update the confidence values. Using Equation (3), the confidence value at node b becomes 0, and at node a 0.44. Now when the threads steal work from the root, the situation has changed. Since one of the most fruitful parts of the left branch has been completely searched without producing a solution, it has become much less likely that there is a solution down the left branch. The updated confidence value reflects this. Now the threads will be distributed such that 4 threads are down the left branch and 4 threads are down the right branch, as shown in Figure 1(b).

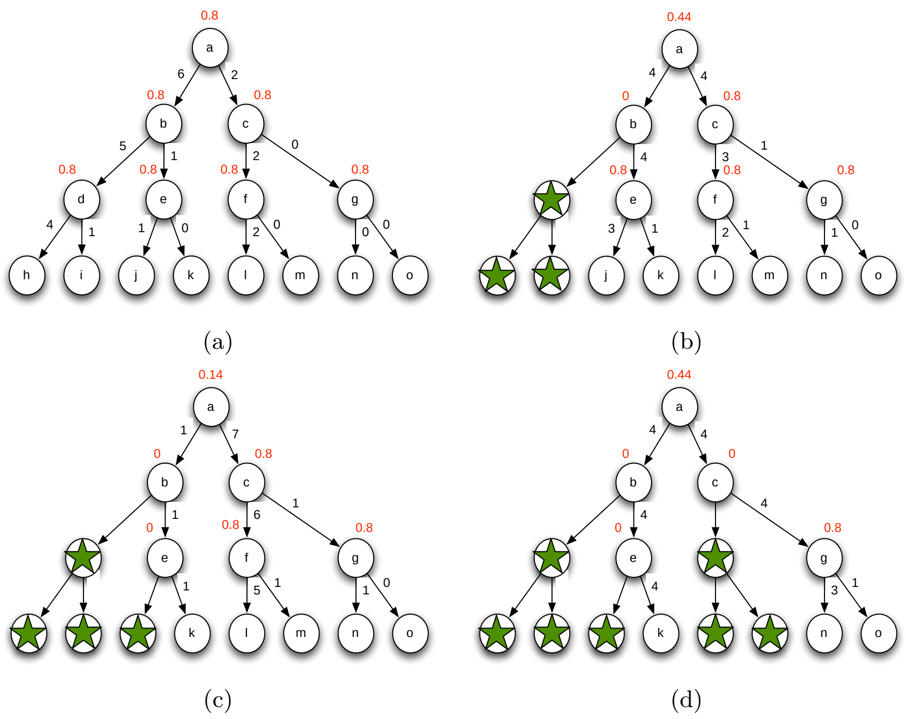

Fig. 1. Example with reasonably high confidence

Next, perhaps the subtree starting at node j finishes. The confidence value at node e then becomes 0, the confidence value at node b remains 0 and the confidence value at node a becomes 0.14. The vast majority of the fruitful places in the left branch has been exhausted without finding a solution, and the confidence value at the root has been updated to strongly favor the right branch. The threads will now be distributed such that 7 threads go down the right and 1 thread goes down the left, as shown in Figure 1(c).

Next, suppose the subtree starting at node f finishes. The confidence value at node c becomes 0 and the confidence value at node a becomes 0.44. Since the most fruitful part of the right branch has also failed, the confidence value now swings back to favor the left branch more, as shown in Figure 1(d). This kind of confidence updating and redistribution of threads will continue, distributing the threads according to the current best solution density estimates. In our explanation here, for simplicity we only updated the confidence values very infrequently. In the actual implementation, confidence values are updated after every job is finished and thus occur much more frequently and in much smaller sized chunks.

Example with low confidence. For the second example, suppose we knew that the heuristic was very bad and was basically random. We may use *conf* $= 0.5$, i.e. the initial solution density estimates down the left and right branch are equal.

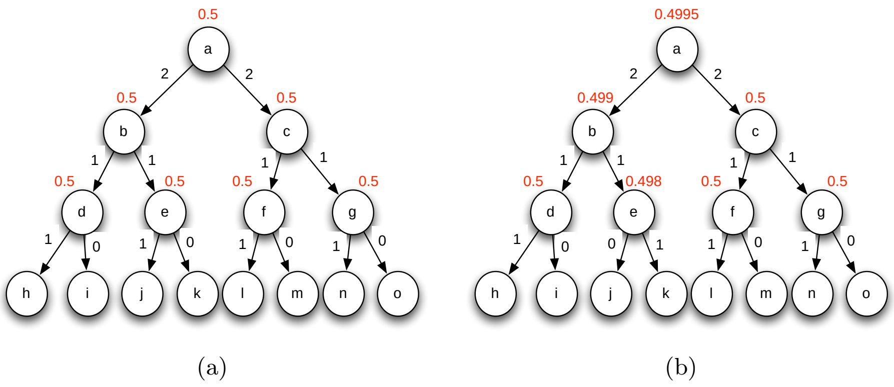

Fig. 2. Example with low confidence

Suppose we have 4 threads. Initially, all the confidence values are 0.5. The threads will distribute as shown in Figure 2(a). This distributes the threads as far away from each other as possible which is exactly what we want. However, if the search tree is deep, and the first few decisions that the each threads makes are wrong, all threads may still get stuck and never find a solution.

This is where the *restart* limit kicks in. After a certain time threshold is reached, the threads abandon their current search and begin work stealing from the root again. Since the confidence values are updated when they abandon their current job, they take a different path when they next steal work. For example, if the thread down node `e` abandons after having finished a subtree with root node at depth 10, then the confidence at node `e` becomes 0.498, the confidence at node `b` becomes 0.499, and the confidence at node `a` becomes 0.4995. Then when a thread steals work from the root, it will again go left, then right. When it gets to node `e` however, the confidence value is 0.498 and there are no threads down either branch, thus it will go right at this node instead of left like last time. The result is shown in Figure 2(b). The updated confidence value has guided the thread to an unexplored part of the search tree that is as different from those already searched as possible. This always happens because solution density estimates are decremented whenever a part of a subtree is found to have failed, so the confidence will always be updated to favour the unexplored parts of the search tree.

Emulating standard search patterns. As some other examples, we briefly mention what confidence models lead to some standard search patterns. DFS: $conf = 1$, $restart = \infty$. IDFS: $conf = 1$, $restart = 1000$. LDS: $conf = 1\text{-}\epsilon$, $restart = 1$ node. DDS: $conf = 0.5$ if depth $<$ d, 1-ϵ if depth $\geq$ d, $restart = 1$ node.

4 Experimental Evaluation

Confidence-based work stealing is implemented using Gecode 3.0.2 [18] with an additional patch to avoid memory management contention during parallel execution. The benchmarks are run on an Mac with 2× 2.8 GHz Xeon Quad Core E5462 processors with 4Gb of memory. However, due to memory limitations, we could not run the large instances of n-Queens or Knights on this machine. We run those two benchmarks on a Dell PowerEdge 6850 with 4× 3.0 GHz Xeon Dual Core Pro 7120 CPUs with 32Gb of memory. 8 threads are used for the parallel search algorithm. We use a time limit of 20 min CPU time (2.5 min wall clock time for 8 threads), a restart time of 5 seconds, and a dynamic granularity bound that adjusts itself to try to steal no more than once every 0.5 seconds. We collected the following data: wall clock runtime, CPU utilization, communication overhead, number of steals, total number of nodes searched and number of nodes explored to find the optimal solution.

Optimization problems. In our first set of experiments we examine the efficiency of our algorithm for three optimization problems from Gecode's example problems. The problems are: Traveling Salesman Problem (TSP), Golomb-Ruler and Queens-Armies. A description of these problems can be found at [18]. We use the given search heuristic (in the Gecode example file) for each, except for TSP where we try both a strong heuristic based on maximising cost reduction and a weak heuristic that just picks variables and values in order. For TSP, we randomly generated many instances of an appropriate size for benchmarking. Only the size 12 and size 13 instances of Golomb Ruler, and only the size 9 and size 10 instances of Queen-Armies, are of an appropriate size for benchmarking. We use the simple confidence model with $conf$ = 1, 0.66 and 0.5. The results are given in Table 1.

It is clear that in all of our problems, runtime is essentially proportional to the number of nodes searched, and it is highly correlated to the amount of time taken to find the optimal solution. The quicker the optimal solution is found, the fewer the nodes searched and the lower the total runtime. The communication cost, which includes all work stealing and synchronisation overheads, is less than 1% for all problems.

The strong heuristic in TSP is quite strong. Using $conf$ = 1 achieves near perfect algorithmic efficiency, where algorithm efficiency is defined as the total number of nodes searched in the parallel algorithm vs the sequential algorithm. Other values of $conf$ clearly cause an algorithmic slowdown. The optimal solution is found on average 3.2 and 3.3 times slower for $conf$ = 0.66 and 0.5 respectively, resulting in an algorithmic efficiency of 0.81 and 0.80 respectively. The opposite is true when the weak heuristic is used. Using $conf$ = 1 or 0.66 allows us to find the leftmost optimal solution in approximately the same number of nodes as the sequential algorithm, but using $conf$ = 0.5 to reflect that the heuristic is weak allows the algorithm to find the optimal solution even faster, producing an algorithmic efficiency of 1.14.

The branching heuristic in Golomb Ruler is a greedy heuristic that selects the minimum possible value for the variable at each stage. This is a reasonable heuristic but by no means perfect. It turns out that for Golomb Ruler 12 and 13, the optimal solution does not lie directly in the left-most branch, and a certain degree of non-greediness leads to super-linear solution finding efficiencies.

Table 1. Experimental results for optimization problems with simple confidence model. The results show: wall clock runtime in seconds (Runtime), speedup relative to the sequential version (Speedup), and runtime efficiency (RunE) which is Speedup/8, CPU utilization (CPU%), communication overhead (Comm%), number of steals (Steals), total number of nodes explored (Nodes), the algorithmic efficiency (AlgE) the total number of nodes explored in the parallel version versus the sequential version, the number of nodes explored to find the optimal solution (Onodes), and the solution finding efficiency (SFE) the total number of nodes explored in the parallel version to find the optimal versus the sequential version. Values for Runtime, CPU%, Comm%, Steals, Nodes, and Onodes are the geometric mean of the instances solved by all 4 versions.

TSP with strong heuristic, 100 instances (Mac)

conf	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	313.3	—	—	100.0%	0.0%	—	5422k	—	1572k	—
1	38.2	7.25	0.91	96.5%	0.4%	708	5357k	1.01	1589k	0.99
0.66	47.2	5.88	0.74	93.7%	0.1%	319	6657k	0.81	5130k	0.31
0.5	48.0	5.77	0.72	92.9%	0.1%	467	6747k	0.80	5275k	0.30

TSP with weak heuristic, 100 instances (Mac)

conf	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	347.8	—	—	99.0%	0.0%	—	7.22M	—	1.15M	—
1	46.7	7.45	0.93	99.4%	0.6%	1044	6.96M	1.04	1.09M	1.06
0.66	45.8	7.60	0.95	96.9%	0.1%	379	7.02M	1.03	1.10M	1.05
0.5	41.6	8.36	1.05	97.5%	0.1%	304	6.36M	1.14	0.96M	1.20

Golomb Ruler, 2 instances (n = 12, 13) (Mac)

conf	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	562	—	—	100.0%	0.0%	—	9.71M	—	1.07M	—
1	69.0	8.15	1.02	99.3%	0.2%	572	8.96M	1.08	0.81M	1.33
0.66	59.0	9.54	1.19	99.3%	0.1%	346	7.58M	1.28	0.49M	2.21
0.5	65.2	8.63	1.08	99.3%	0.1%	259	8.42M	1.15	0.66M	1.63

Queen Armies, 2 instances (n = 9, 10) (Mac)

conf	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	602	—	—	100.0%	0.0%	—	13.6M	—	845k	—
1	87.1	6.91	0.86	99.3%	0.7%	1521	14.5M	0.94	1878k	0.45
0.66	86.3	6.98	0.87	98.8%	0.2%	1143	14.5M	0.96	2687k	0.31
0.5	86.0	7.00	0.87	99.5%	0.2%	983	14.5M	0.95	2816k	0.30

The results for Queens-Armies show little difference depending on confidence. Clearly the heuristic is better than random at finding an optimal solution, and solution finding efficiency degrades slightly as we ignore the heuristic. But the overall nodes searched are almost identical for all confidence values, as the work required for the proof of optimality make up the bulk of the run time, and the proof of optimality parallelises trivially regardless of confidence.

Satisfaction problems. In our second set of experiments we examine the efficiency of our algorithm for three satisfaction problems from Gecode's examples [18]. The problems are: n-Queens, Knights, and Perfect-Square.

Table 2. Experimental results for satisfaction problems with simple confidence model

n-Queens, 100 instances (n = 1500, 1520, ..., 3480)

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE
Seq	4	2.9	—	—	99.9%	0.0%	—	1859	—
1	4	10.4	—	—	99.0%	86.6%	2	1845	—
0.66	29	18.0	—	—	81.6%	0.3%	9	15108	—
0.5	100	2.9	—	—	65.5%	1.6%	8	14484	—

Knights, 40 instances (n = 20, 22, ..., 98)

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE
Seq	7	0.22	—	—	99.9%	0.0%	—	1212	—-
1	7	0.26	—	—	68.1%	59.7%	2	1150	—
0.66	13	0.50	—	—	48.0%	4.7%	8	8734	—
0.5	21	0.66	—	—	35.2%	6.0%	8	8549	—

Perfect-Square, 100 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE
Seq	15	483.1	—	—	99.9%	0.0%	—	213k	—
1	13	72.3	6.68	0.83	98.0%	19.1%	419	216k	0.99
0.66	14	71.2	6.78	0.85	86.4%	2.9%	397	218k	0.98
0.5	82	8.9	54.02	6.75	89.0%	4.8%	21	32k	6.64

The sequential version solved very few instances of n-Queens and Knights. Furthermore, all those solves are extremely fast ($<$ 3 sec) and are caused by the search engine finding a solution at the very leftmost part of the search tree. Most of the time spent in those runs is from moving down to the leaf of the search tree rather than actual search and is not parallelisable, thus comparison of the statistics for the parallel vs sequential algorithms on those instances is not meaningful as there is very little work to parallelize. The number of instances solved is the more interesting statistic and is a better means of comparison. The parallel algorithm beats the sequential algorithm by an extremely large margin in terms of the number of instances solved.

n-Queens and Knights both have very deep subtrees and thus once the sequential algorithm fails to find a solution in the leftmost subtree, it will often end up stuck effectively forever. Modeling the fact that the branching heuristic is very weak at the top by using $conf = 0.5$ clearly produce a super linear speedup. The parallel algorithm solves 100 out of 100 instances of n-Queens compared to 4 out of 100 instances for the sequential algorithm or the parallel algorithm with $conf = 1$. The speedup cannot be measured as the sequential algorithm does not terminate for days when it fails to find a solution quickly. Similarly the parallel algorithm with $conf = 0.5$ solved 21 instances of Knights compared to 7 for the sequential and the parallel version with $conf = 1$.

Perfect Square's heuristic is better than random, but is still terribly weak. Using $conf = 0.5$ to model this once again produces super linear speedup, solving 82 instances out of 100 compared to 15 out of 100 for the sequential algorithm. We can compare run times for this problem as the sequential version solved a

Table 3. Experimental results showing Nodes and algorithmic efficiency (AlgE) using accurate confidence values, where we follow the confidence value to degree α.

	Seq		$\alpha = 1$		$\alpha = 0.5$		$\alpha = 0$		$\alpha = -0.5$		$\alpha = -1$	
Golomb-Ruler 12	5.31M	—	2.24M	2.37	3.48M	1.53	4.27M	1.24	10.8M	0.49	10.6M	0.50
Golomb-Ruler 13	71.0M	—	53.2M	1.34	57.6M	1.23	61.9M	1.15	74.8M	0.95	111M	0.64

fair number of instances and those solves actually require some work (483 sec on average). The speedup in this case is 54 using 8 threads.

Using accurate confidence values. So far, we have tested the efficiency of our algorithm using simple confidence models where the confidence value is the same for all nodes. This does not really illustrate the algorithm's full power. We expect that it should perform even better when node-specific confidence values are provided, so that we can actually encode and utilise information like, the heuristic is confident at this node but not confident at that node, etc. In our third set of experiments, we examine the efficiency of our algorithm when node specific confidence values are provided.

Due to our lack of domain knowledge, we will not attempt to write a highly accurate confidence heuristic. Rather, we will simulate one by first performing an initial full search of the search tree to find all solutions, then produce confidence estimates for the top few levels of the search tree using several strategies like, follow the measured solution density exactly, follow it approximately, ignore it, go against it, etc, to see what effect this has on runtime. Let α quantify how closely we follow the measured confidence value and let *conf* be the measured confidence value. Then we use the following formula for our confidence estimate: $conf' = \alpha \times conf + (1-\alpha) \times 0.5$. If $\alpha = 1$, then we follow it exactly. If $\alpha = -1$, we go against it completely, etc. We use the Golomb-Ruler problem for our experiment as the full search tree is small enough to enumerate completely. The results are shown in Table 3.

The results show that using confidence values that are even a little biased towards the real value is sufficient to produce super linear speedup. And not surprisingly, going against the real value will result in substantial slowdowns.

5 Conclusion

By analysing work stealing schemes using a model based on solution density, we were able to quantitatively relate the strength of the branching heuristic with the optimal place to steal work from. This leads to an adaptive work stealing algorithm that can utilise confidence estimates to automatically produce "optimal" work stealing patterns. The algorithm produced near perfect or better than perfect algorithmic efficiency on all the problems we tested. In particular, by adapting to a steal high, interleaving search pattern, it is capable of producing super linear speedup on several problem classes. The real efficiency is lower than the algorithmic efficiency due to hardware effects, but is still quite good at a speedup of at least 7 at 8 threads. Communication costs are negligible on all problems even at 8 threads.

Acknowledgments. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

1. Michel, L., See, A., Van Hentenryck, P.: Parallelizing constraint programs transparently. [19] 514–528
2. Perron, L.: Search procedures and parallelism in constraint programming. In Jaffar, J., ed.: Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming. Volume 1713 of Lecture Notes in Computer Science., Alexandra, VA, USA, Springer-Verlag (October 1999) 346–360
3. Schulte, C.: Parallel search made simple. In Beldiceanu, N., Harvey, W., Henz, M., Laburthe, F., Monfroy, E., Müller, T., Perron, L., Schulte, C., eds.: Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000. Number TRA9/00, 55 Science Drive 2, Singapore 117599 (September 2000) 41–57
4. Rolf, C.C., Kuchcinski, K.: Load-balancing methods for parallel and distributed constraint solving. In: IEEE International Conference on Cluster Computing. (October 2008) 304–309
5. Kumar, V., Rao, V.N.: Parallel depth first search. Part II. Analysis. International Journal of Parallel Programming **16**(6) (1987) 501–519
6. Caseau, Y., Laburthe, F.: Solving small TSPs with constraints. In Naish, L., ed.: Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, The MIT Press (July 1997) 316–330
7. Véron, A., Schuerman, K., Reeve, M., Li, L.L.: Why and how in the ElipSys OR-parallel CLP system. In Bode, A., Reeve, M., Wolf, G., eds.: PARLE. Volume 694 of Lecture Notes in Computer Science., Springer (1993) 291–303
8. Rao, V.N., Kumar, V.: Superlinear speedup in parallel state-space search. In Nori, K.V., Kumar, S., eds.: FSTTCS. Volume 338 of Lecture Notes in Computer Science., Springer (1988) 161–174
9. Gendron, B., Crainic, T.G.: Parallel branch-and-bound algorithms: Survey and synthesis. Operations Research **42**(6) (November-December 1994) 1042–1066
10. Quinn, M.J.: Analysis and implementation of branch-and bound algorithms on a hypercube multicomputer. IEEE Trans. Computers **39**(3) (1990) 384–387
11. Mohan, J.: Performance of Parallel Programs: Model and Analyses. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, USA (1984)
12. Meseguer, P.: Interleaved depth-first search. [20] 1382–1387
13. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In Mellish, C.S., ed.: Fourteenth International Joint Conference on Artificial Intelligence, Montréal, Québec, Canada, Morgan Kaufmann Publishers (August 1995) 607–615
14. Walsh, T.: Depth-bounded discrepancy search. [20] 1388–1395
15. Refalo, P.: Impact-based search strategies for constraint programming. In Wallace, M., ed.: CP. Volume 3258 of Lecture Notes in Computer Science., Springer (2004) 557–571
16. Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics. [19] 743–757
17. Szymanek, R., O'Sullivan, B.: Guiding search using constraint-level advice. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: ECAI, IOS Press (2006) 158–162

18. Gecode Team: Gecode: Generic constraint development environment (2006) Available from `http://www.gecode.org`.
19. Bessière, C., ed.: Thirteenth International Conference on Principles and Practice of Constraint Programming. Volume 4741 of Lecture Notes in Computer Science. Springer-Verlag, Providence, RI, USA (September 2007)
20. Pollack, M.E., ed.: Fifteenth International Joint Conference on Artificial Intelligence. Morgan Kaufmann Publishers, Nagoya, Japan (August 1997)