# Confidence based Work Stealing in Parallel Constraint Programming

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

[1] NICTA 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.** In parallel constraint solving, 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. In this paper we examine quantitatively how optimal work stealing can be performed given an estimate of the relative solution densities of the subtrees at each node in the search tree and show how this is related to the branching heuristic strength. We propose an adaptive work stealing algorithm that automatically performs different work stealing strategies based on the strength of the branching heuristic at each node. Many parallel depth-first search patterns arise naturally from our algorithm. Our algorithm is able to produce near perfect or super linear algorithmic efficiencies on all problems tested. Real speedups using 8 threads ranges from 4-5 times speedup to super linear speedup.

## 1 Introduction

In parallel constraint solving, work stealing has often been seen only as a mechanism for keeping processors occupied. Analysis of work stealing schemes often assume that the amount of work to be done is fixed and independent of the work stealing scheme, e.g. [1]. While this is true for certain kinds of problems, e.g. finding all solutions, proving unsatisfiability, it is not true for others, e.g. finding the first solution, finding the optimal solution. Such analyses fail to account for the fact that the place from which work is stolen determines the search strategy and can have a dramatic effect on the efficiency of the parallel algorithm. Many systems choose to steal from as close to the root of the search tree as possible, e.g. [2], as this tends to give the greatest granularity. However, this is not always the best place to steal from in terms of the efficiency of the algorithm.

We illustrate how work stealing from different places can have different effects on efficiency with two examples. 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 pick an unexplored part of the search tree and steal that subtree off its current owner. This continues until a solution is found, or the entire search tree has been searched in the case of unsatisfiability or optimization.

*Example 1.* The first problem we will consider is the Travelling Salesman Problem. 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 ∼30% more nodes on average (see Table 1).

The explanation for this is simple. Let us examine the details of one particular instance. In this instance, with the sequential algorithm, the optimal solution is found after 47 seconds of CPU time, after which the algorithm spends another ∼300 seconds proving that no better solution exists. When work stealing is done as left and low as possible in the parallel search, all of the threads are working towards finding that leftmost optimal solution, and the optimal solution is found in 47 seconds of total CPU time as before (wall clock time ∼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.

If we steal high however, only 1 of the threads is actually exploring the leftmost part of the search tree and working towards that leftmost optimal solution. The other 7 threads are off searching 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 that no better solution exists. What has happened is that we got no speedup whatsoever for finding the optimal solution, but linear speedup for proving that no better solution exists. Since we found the optimal solution so much later in the search (376 seconds CPU time instead of 47 seconds), the threads spent an enormous amount of CPU time searching without the pruning benefits of the optimal solution, thus the total number of nodes searched in this instance is dramatically increased, leading to a great loss of efficiency. Clearly, this effect gets worse as a higher number of threads is used. □

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.* The second problem we will consider is the $n$-Queens problem. The search tree is very deep and a top level mistake will not be recovered from for hours. Stealing low in parallel search solves the instance within the time limit if and only if the sequential depth first search solved it within the time limit. This only occurred when a solution falls in the very leftmost part of the search tree (only 4 instances out of 100 tested, 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* [3] claims 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 [4] (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. we have no idea how the solutions

are distributed), but is not valid in systems where the 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 [5] 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. [6, 7]).

Our contributions in this paper are as follows. We perform a quantitative analysis of how different work stealing strategies affect the total amount of work performed and explain the relationship between branching heuristic strength and the optimal search strategy. We propose an adaptive work stealing algorithm that, when provided with a user given *confidence*, which is the estimated ratio of solution densities between the left and right subtrees at each node, will work steal in a near optimal manner. We show that confidence based work stealing leads to very good algorithmic efficiencies, i.e. it does not visit many more nodes, and sometimes much less, than sequential DFS (Depth First Search).

Although our analysis is done in the context of work stealing in parallel constraint programming systems, the analysis is actually about the relationship between branching heuristic strength and the optimal search order in the search tree created by that branching heuristic. Thus the analysis actually applies to all complete tree search algorithms whether sequential or parallel. As we will show later, when the assumptions about branching heuristic strength that lie behind standard sequential algorithms such as DFS, Interleaved Depth First Search (IDFS), Limited Discrepancy Search (LDS) or Depth-bounded Discrepancy Search (DDS) is given to our algorithm as *confidence* estimates, our algorithm automatically produces the exact same search patterns used in those algorithms. Thus our 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. Our algorithm is also fully parallel and thus we have automatically parallelised DFS, IDFS, LDS and DDS as well.

The layout of the paper is as follows. In section 2 we perform a quantitative analysis of optimal work stealing. In section 3 we describe our adaptive work stealing algorithm. In section 4 we give examples of the behaviour of our algorithm. In section 5 we present our experimental evaluation. Finally in section 6 we conclude.

## 2 Analysis of Work Allocation

In this section we show quantitatively that the strength of the branching heuristic determines the optimal place to work steal 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 system of constraints $C$ over variables $V$ with initial domain $D$ assigning possible values $D(v)$ to

each variables $v \in V$. 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 it cannot detect any new values that 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 ($|D(v)| = 1, v \in V$) then the resulting state is a *solution state*. Failure states 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* $\vee_{i=1}^{n} c_i$ which is consequence of the current state $C \wedge D$. The child states of this state 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 $x$ nodes and $y$ solution state nodes is $y/x$. The solution density is the inverse of the *mean nodes to solution* of $T$ defined as $x/y$.

*Optimal split for binary nodes* For simplicity, assume that the cost of visiting each node in the search tree is roughly equal. Intuitively, the optimal way to perform a search is to assign all of our threads to the most promising parts of the search tree at each stage. These places are the parts of the search tree where the mean nodes to solution is lowest, or in other words where the solution density is highest. Assuming an oracle that could give us accurate solution density information, 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, thus we have to take various other factors into account. Namely:

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 subtrees that are close together, as they share decision constraints from higher up in the tree, and these constraints may already have made solutions impossible or plentiful.
3. The solution density estimate of a subtree should decrease as nodes in that tree are examined without finding a solution. This is caused by two factors.
   (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 work steal.

*Example 3.* Let $T$ be a search tree with a binary decision at the root. Let $A = 0.6$ and $B = 0.4$ be the solution density estimates for the left and right branches of $T$. Assume also that the two subtrees have the same number of nodes. If we

had 8 threads, what is the optimal division of threads between the two branches such that the expected time to find a solution is lowest?

If the solution density estimate was perfect, we would simply send all 8 threads down the left branch. However, according to (1) the estimate has a very large error. Further according to (2) the solution density of the subtrees down the left branch are highly correlated. If we send all 8 threads down the left branch, and it turns out that the real solution density is small or zero, then all 8 threads end up stuck. Because of (1) and (2), it is actually better to send some of the threads down the right branch as well as long as $B$ is not far smaller than $A$, for example, for values of $A = 0.6, B = 0.4$, we may wish to send 6 threads down the left branch and 2 threads down the right branch. □

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, i.e. 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 (see Appendix A for the details of the calculation):

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

The shape of this function does not depend on the absolute values of $A$ and $B$ (which only serves 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 this function for a given value of $r$ is shown in Figure 1. This graph tells us the optimal way to divide up our processing power so that we have the lowest expected number of nodes to search.

As can be seen, although not linear, the optimal values of $p$ are 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. Or if $r = 0.5$, which means that $A = B$, then it is near optimal to send equal amounts of processing power down the two branches.

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, or 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 everytime. This is equivalent to stealing as left and as low as possible. The

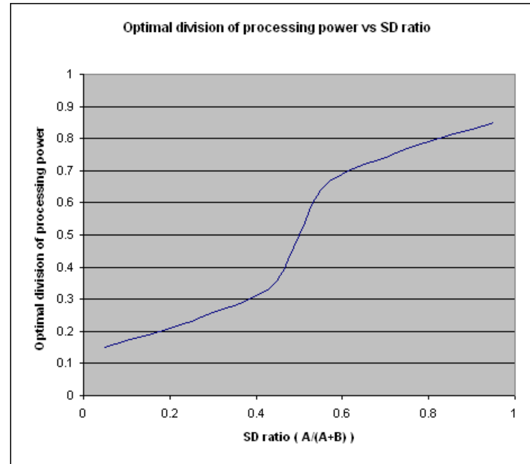


**Fig. 1.** Optimal division of processing power based on solution density ratio

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 clue 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 as high as possible.[3]

## 3 Adaptive work stealing

Our analysis shows that the optimal work stealing strategy is dependant 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 in Gecode [8].

### 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) $x_i$. If $S$ and $A_i$

[3] We ignore the possibility of an *anti-heuristic* where the right branch is preferable to the left.

are estimates of average solution density, then clearly: $S = \sum_{i=1}^{n} A_i x_i / \sum_{i=1}^{n} x_i$, i.e. the average solution density of the subtree is the weighted average of the solution densities of its child subtrees.

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

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 root 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 $\rho$ represents the effect on the solution density due to the constraint added at the root node, and $A'_i$ represents the effect on the solution density due to constraints added within branch $i$. Then $\rho$ is a common factor in the solution density estimates for each branch and represents the correlation between them. We have that:

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

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

$$S = \frac{\sum_{i=1}^{n} A_i}{n} \tag{2}$$

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.

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 $i$th levels above the root of the failed subtree and $\bar{r}_i = 1 - r_i, \bar{r}'_i = 1 - r'_i$ otherwise. Then:

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

### 3.2 Confidence model

Given a confidence at each node, we now know how to work steal "optimally", and how to update confidences as search proceeds. But how do we get an initial confidence at each node. Ideally, the problem modeller, with expert knowledge about the problem and the branching heuristic can develop a solution density heuristic that gives us 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.

### 3.3 The 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 a job is finished, the confidence values of all nodes above the finished subtree is updated as described in (3).

When work stealing is required, we start at the root of the tree, and use the number of threads down each branch, the confidence value, and the optimal division derived in Section 2 to work out whether the thread should be assigned to the left branch or the right branch. We then move on to that node and repeat. We continue 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 job 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, as this could take exponential time [9]. Instead, after a given *restart* time has passed, the thread returns the unexplored parts of its subtree to the master and work steals again from the top. This is similar to the idea used in interleaving DFS [10].

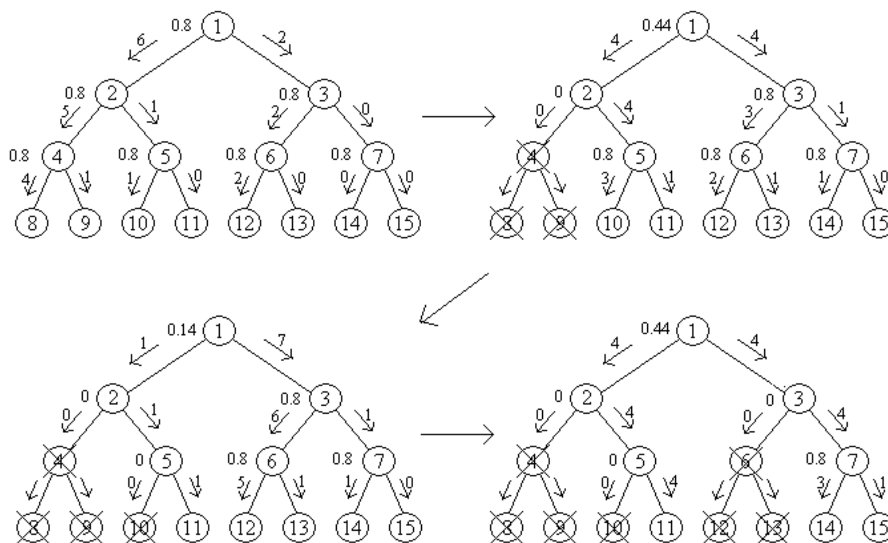

**Fig. 2.** Example 1

## 4 Sample behaviour of adaptive work stealing algorithm

In this section, we go through some examples of how the work stealing and confidence updating works in our algorithm. For the first example, suppose we know that the branching heuristic is reasonably strong, but not perfect. We may use $conf = 0.8$. Refer to Figure 2 in the explanation.

Let's suppose we have 8 threads. Initially, all the confidence values are 0.8. When the 8 threads attempt to work steal at the root, the first thread will go down the left hand side. The second thread will go down the left hand side as well. The 3rd thread will go down the right hand side. The fourth thread will go down the left hand side, etc, until we end up with 6 threads down the left and 2 threads down the right. At node 2, we will have 5 threads down the left and 1 thread down the right. At node 3, we will have 2 threads down the left, 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.

Suppose as search progresses the subtree starting at node 4 finishes without producing a solution. Then we need to update the confidence values. Using (3), the confidence value at node 2 becomes 0, and the confidence value at node 1 becomes 0.44. Now when the threads work steal from the root, things are different. 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. Next, perhaps the subtree starting at node 10 finishes. The confidence value at node 5 then becomes 0, the confidence value at node 2 remains 0 and the confidence value at node 1 becomes 0.14.

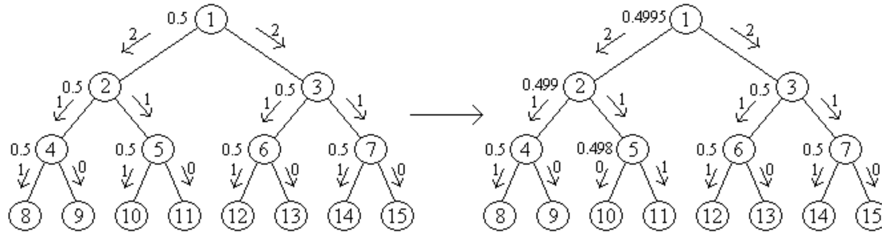


**Fig. 3.** Example 2

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 go down the left. Next, suppose the subtree starting at node 6 finishes. The confidence value at node 3 becomes 0 and the confidence value at node 1 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. This kind of confidence updating and redistribution of threads will continue on, 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.

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. Refer to Figure 3 in the explanation.

Let's suppose we have 4 threads. Initially, all the confidence values are 0.5. When the 4 threads attempt to work steal at the root, the first thread will go left, then left, then left, etc. The second thread will go right, then left, then left, etc. The third thread will go left, then right, then left, etc, and the fourth thread will go right, then right, then left, etc. 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 threads made within its own subtree are wrong, they may still all get stuck and never find a solution. This is where the interleaving 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 work steal. For example, if the thread down node 5 abandons after having finished a subtree with root node at depth 10, then the confidence at node 5 becomes 0.498, the confidence at node 2 become 0.499, and the confidence at node 1 becomes 0.4995. Then when the thread work steals from the root, it will again go left, then right. When it gets to node 5 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 updated confidence

values 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 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.

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

## 5 Experimental evaluation

The confidence based work stealing is implemented in Gecode 2.1.1 [8]. The benchmarks are run on a Dell PowerEdge 6850 with 4x 3.0 Ghz Xeon Dual Core Pro 7120 CPUs. 8 threads are used for the parallel search algorithm. We use a time limit of 20 min CPU time (so 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.

In our first set of experiments we examine the efficiency of our algorithm for two optimization problems from Gecode's example problems. The problems are: Travelling Salesman Problem (TSP), Photo and Queens-Armies. A description of these problems can be found at [8]. 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 both Photo and TSP, we randomly generated many instances of an appropriate size for benchmarking. 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 apparent from our experiments that the hardware/OS we experimented on is highly non-ideal and does not in fact give us a linear increase in real processing speed when more processors are used. We suspect this is due to issues such as cache contention, memory contention, context switching, etc. The effect causes threads to slow down by up to 40% at 8 threads. In view of this, the primary statistics we will look at in our analysis of our algorithm will be algorithmic efficiency and the communication cost. Algorithmic efficiency minus the communication cost represents the theoretical efficiency on an ideal parallel computer. The runtime efficiency represents what you may get on a real world, non-ideal parallel computer.

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

**Table 1.** Experimental results for optimization problems with simple confidence model. The results show: number of problems solved in the time limit (Solved), 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, 200 instances

| *conf* | Solved | Runtime | Speedup | RunE | CPU% | Comm% | Steals | Nodes | AlgE | Onodes | SFE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Seq | 181 | 56.0 | — | — | 99.8% | 0.0% | — | 1240k | — | 180k | — |
| 1 | 172 | 11.3 | 4.95 | 0.62 | 94.7% | 2.8% | 447 | 1222k | 1.01 | 218k | 0.82 |
| 0.66 | 170 | 13.3 | 4.20 | 0.53 | 94.6% | 0.5% | 370 | 1517k | 0.82 | 580k | 0.31 |
| 0.5 | 160 | 16.2 | 3.45 | 0.43 | 94.2% | 1.3% | 533 | 1564k | 0.80 | 658k | 0.27 |

TSP with weak heuristic, 200 instances

| *conf* | Solved | Runtime | Speedup | RunE | CPU% | Comm% | Steals | Nodes | AlgE | Onodes | SFE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Seq | 189 | 78.6 | — | — | 99.8% | 0.0% | — | 1.99M | — | 1.59M | — |
| 1 | 186 | 17.7 | 4.45 | 0.56 | 96.5% | 4.0% | 686 | 1.99M | 1.00 | 1.59M | 1.00 |
| 0.66 | 186 | 17.7 | 4.46 | 0.56 | 96.3% | 0.4% | 319 | 1.97M | 1.01 | 1.60M | 1.00 |
| 0.5 | 184 | 15.7 | 5.01 | 0.63 | 95.5% | 0.8% | 287 | 1.73M | 1.15 | 1.39M | 1.15 |

Photo, 200 instances

| *conf* | Solved | Runtime | Speedup | RunE | CPU% | Comm% | Steals | Nodes | AlgE | Onodes | SFE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Seq | 173 | 63.9 | — | — | 99.9% | 0.0% | — | 5.01M | — | 622k | — |
| 1 | 152 | 15.5 | 4.12 | 0.52 | 98.0% | 1.7% | 636 | 4.93M | 1.02 | 542k | 1.15 |
| 0.66 | 153 | 15.5 | 4.12 | 0.52 | 97.5% | 0.4% | 388 | 4.91M | 1.02 | 467k | 1.33 |
| 0.5 | 152 | 15.4 | 4.15 | 0.52 | 97.7% | 0.4% | 253 | 4.90M | 1.02 | 492k | 1.26 |

Queen Armies, 2 instances

| *conf* | Solved | Runtime | Speedup | RunE | CPU% | Comm% | Steals | Nodes | AlgE | Onodes | SFE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Seq | 2 | 1146 | — | — | 99.7% | 0.0% | — | 27.1M | — | 800k | — |
| 1 | 2 | 219 | 5.24 | 0.65 | 98.7% | 1.1% | 2519 | 28.8M | 0.94 | 1669k | 0.48 |
| 0.66 | 2 | 213 | 5.38 | 0.67 | 98.2% | 0.5% | 1924 | 28.4M | 0.96 | 1781k | 0.45 |
| 0.5 | 2 | 217 | 5.29 | 0.66 | 98.3% | 0.4% | 1631 | 28.6M | 0.95 | 1902k | 0.42 |

1% for most problems, but goes up to around 3-4% for some steal low strategies. For algorithmic efficiency, we will examine each the problem in turn.

The strong heuristic in TSP is quite strong. Using $conf = 1$ achieves near perfect algorithmic efficiency. Other values of *conf* clearly cause an algorithmic slowdown. The optimal solution is found on average 2.7 and 3.0 times slower for $conf = 0.66$ and 0.5 respectively, resulting in an algorithmic efficiency of 0.82 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 1.15 compared to the sequential algorithm.

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

*n*-Queens, 100 instances

| *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

| *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 branching heuristic in Photo is designed to minimize the size of the search tree, rather than to place the solutions on the left side of the tree, hence, it is a "weak" heuristic as far as our analysis is concerned. Using $conf = 0.66$ and 0.5 to reflect this clearly produce higher solution finding efficiency than $conf = 1$, giving 1.33 and 1.26 vs 1.15 respectively. However, for the Photo problem, there are so many optimal solutions in the search tree that one gets found extremely quickly regardless of which strategy is used, and hence finding the optimal solution faster has no real effect on total runtime.

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.

In our second set of experiments we examine the efficiency of our algorithm for three satisfaction problems from Gecode's examples [8]. The problems are: *n*-Queens, Knights, and Perfect-Square.

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 travelling down to the leaf of the search tree rather than actual search and is not parallelizable, 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. Modelling the fact that the branching heuristic is

**Table 3.** Experimental results using accurate confidence values, where we follow the confidence value to degree $\alpha$.

Golomb-Ruler 12

| $\alpha$ | Nodes | AlgE |
|---|---|---|
| Seq | 5.31M | — |
| 1 | 2.24M | 2.37 |
| 0.5 | 3.48M | 1.53 |
| 0 | 4.27M | 1.24 |
| -0.5 | 10.8M | 0.49 |
| -1 | 10.6M | 0.50 |

Golomb-Ruler 13

| $\alpha$ | Nodes | AlgE |
|---|---|---|
| Seq | 71.0M | — |
| 1 | 53.2M | 1.34 |
| 0.5 | 57.6M | 1.23 |
| 0 | 61.9M | 1.15 |
| -0.5 | 74.8M | 0.95 |
| -1 | 111M | 0.64 |

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 runtimes for this problem as the sequential version solved a 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.

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 is the most primitive way to use our algorithm and does not really illustrate its full power. We expect that our algorithm should perform even better when confidence values specific to each node 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 $\alpha$ 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 (see [8]) 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.

## 6 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 work steal 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 4-5 at 8 threads. Communication costs are negligible on all problems even at 8 threads.

## References

1. Kumar, V., Rao, V. N.: Parallel depth first search. Part II. Analysis. In: International Journal of Parallel Programming, 16(6):501–519, (1987)
2. Schulte, C.: Parallel search made simple. In: Beldiceanu, N., Harvey, W., Henz, M., Laburthe, F., Monfroy, E., Müller, T. (eds.) Proceedings of TRICS: Techniques for Implementing Constraint Programming Systems, a post-conference workshop of CP 2000, Singapore, September (2000)
3. Veron, A., Schuerman, K., Reeve, M., Li, L.: Why and how in the ElipSys Or-Parallel CLP system. In: Proceedings of PARLE'93, pp. 291–302. Springer-Verlag (1993)
4. Rao, V. N., Kumar, V.: Superlinear Speedup in State-Space Search, Technical Report, AI Lab TR88-80, University of Texas at Austin June (1988)
5. Gendron, B., Crainic, T.G.: Parallel branch-and-bound algorithms: Survey and Synthesis. In: Operations Research 42, pp. 1042–1066. (1994)
6. Quinn, M. J.: Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer. In: IEEE Trans. Comp. 39(3), pp. 384–387 (1990)
7. Mohan, J.: Performance of Parallel Programs: Model and Analyses. Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Penn. (1984)
8. Gecode. `www.gecode.org`
9. Perron, L.: Search procedures and parallelism in constraint programming. In: Jaffar, J., editor, Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (1999)
10. Meseguer, P.: Interleaved depth-first search. In: International Joint Conference on Artificial Intelligence, volume 2, pp. 1382–1387, August (1997)

## A Expected Search Time Calculation

Calculating the expected search time requires a double integration of a hybrid function. Since this is rather difficult we will pick a simple solution density probability distribution. Suppose the solution density probability distribution is uniform, i.e. when the solution density estimate is $A$, there is an equal chance of the actual value being anything from 0 to $A$. The real solution density values are actually discrete as there can only be a whole number of solutions, but for ease of integration we leave it as a continuous function. This probability distribution satisfies our criteria that there is a large error in the estimate and that there is a substantial chance that it is actually 0. Suppose the solution density estimates for the left and right branch are $A$ and $B$ respectively, and that they each have $n$ nodes. Suppose also the solutions are randomly distributed among the $n$ nodes. We know that when there are $m$ items randomly located in $n$ locations, the expected number of locations to look before we find one of them is given by $\frac{n+1}{m+1}$. Thus if $a$ is the real solution density in the left branch, the expected time to find a solution in the left branch is $\frac{n+1}{an+1} \approx \frac{n}{an+1} = \frac{1}{a+\frac{1}{n}}$. Suppose we divide up our processing power such that $p$ units is sent down the left branch, and $(1-p)$ units is sent down the right branch. The expected number of nodes searched will depend on which of the branches yield a solution first, thus for real solution density values of $a$ and $b$ for the left and right branch, it is given by the hybrid function:

$$\min\left(\frac{1}{p(a+\frac{1}{n})}, \frac{1}{(1-p)(b+\frac{1}{n})}\right) \tag{4}$$

The expected number of nodes to be searched for solution density estimates $A$ and $B$ for the left and right branch respectively, given a uniform solution density probability distribution will then be given by:

$$\frac{1}{AB}\int_0^A \int_0^B \min\left(\frac{1}{p(a+\frac{1}{n})}, \frac{1}{(1-p)(b+\frac{1}{n})}\right) db\, da \tag{5}$$

To evaluate this, we need to split the integral into two domains corresponding to the two halves of the hybrid function. The boundary of the hybrid function is given by:

$$\begin{aligned} & p(a+\tfrac{1}{n}) = (1-p)(b+\tfrac{1}{n}) \\ \Rightarrow \quad & a = \tfrac{1-p}{p}(b+\tfrac{1}{n}) - \tfrac{1}{n} \\ \text{or} \quad & b = \tfrac{p}{1-p}(a+\tfrac{1}{n}) - \tfrac{1}{n} \end{aligned}$$

There are four cases depending on whether the boundary of the hybrid function intersects the $a$ or the $b$ axis and whether it intersects the $a = A$ line or the $b = B$ line. For $p > 0.5$ and $p(A+\frac{1}{n}) > (1-p)(B+\frac{1}{n})$, which corresponds to intersecting the $b$ axis and the $b = B$ line, we have:

$$\begin{aligned}
&\tfrac{1}{AB} \int_0^A \int_0^B \min(\frac{1}{p(a+\frac{1}{n})}, \frac{1}{(1-p)(b+\frac{1}{n})}) dbda \\
&= \tfrac{1}{AB}[ \int_{\frac{p}{(1-p)n}-\frac{1}{n}}^{B} \int_0^{\frac{1-p}{p}(b+\frac{1}{n})-\frac{1}{n}} \frac{1}{(1-p)(b+\frac{1}{n})} dadb + \\
&\quad \int_0^{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}} \int_0^{\frac{p}{1-p}(a+\frac{1}{n})-\frac{1}{n}} \frac{1}{p(a+\frac{1}{n})} dbda + \\
&\quad \int_{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}^{A} \int_0^B \frac{1}{p(a+\frac{1}{n})} dbda \\
&= \tfrac{1}{AB}[ \int_{\frac{p}{(1-p)n}-\frac{1}{n}}^{B} \frac{1}{p} - \frac{1}{(1-p)n(b+\frac{1}{n})} db + \\
&\quad \int_0^{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}} \frac{1}{1-p} - \frac{1}{pn(a+\frac{1}{n})} da + \\
&\quad \int_{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}^{A} \frac{B}{p(a+\frac{1}{n})} da \\
&= \tfrac{1}{AB}[ [\frac{B}{p} - \frac{1}{(1-p)n} \ln(B+\frac{1}{n}) - \frac{1}{(1-p)n} + \frac{1}{pn} + \frac{1}{(1-p)n} \ln(\frac{p}{(1-p)n})] + \\
&\quad [\frac{B+\frac{1}{n}}{p} - \frac{1}{(1-p)n} - \frac{1}{pn} \ln(\frac{1-p}{p}(B+\frac{1}{n})) + \frac{1}{pn} \ln(\frac{1}{n})] + \\
&\quad [\frac{B}{p} \ln(A+\frac{1}{n}) - \frac{B}{p} \ln(\frac{1-p}{p}(B+\frac{1}{n}))]] \\
&= \tfrac{1}{AB}[ \frac{2(B+\frac{1}{n})}{p} - \frac{1}{n} \ln(\frac{1-p}{p}(nB+1)) + \\
&\quad \frac{B}{p} \ln(\frac{p}{1-p} \frac{(nA+1)}{(nB+1)}) - \frac{2}{(1-p)n} - \frac{1}{(1-p)n} \ln(\frac{1-p}{p}(Bn+1))]
\end{aligned}$$

If we are reasonably high up in the search tree, which is where the results of this calculation is most important, then we can assume that we are expecting a potentially large number of solutions down each branch, e.g. $An, Bn \gg 1$. In that case, all of the terms containing $1/n$ are much smaller than the terms containing $A$ or $B$ and the expression simplifies to:

$$
\begin{aligned}
& \frac{1}{AB}[2\frac{B+\frac{1}{n}}{p} - \frac{1}{n}\ln(\frac{1-p}{p}(nB+1)) + \qquad (6)\\
& \frac{B}{p}\ln(\frac{p}{1-p}\frac{(nA+1)}{(nB+1)}) - \frac{2}{(1-p)n} - \frac{1}{(1-p)n}\ln(\frac{1-p}{p}(Bn+1))] \\
= & \frac{1}{AB}[\frac{2B}{p} + \frac{B}{p}\ln(\frac{pA}{(1-p)B})] \\
= & \frac{1}{pA}(2+\ln(\frac{pA}{(1-p)B})) \qquad (7)
\end{aligned}
$$

The calculation for the case $p < 0.5$ and $p(A+\frac{1}{n}) > (1-p)(B+\frac{1}{n})$ is similar, and after the simplification, yields the same equation as (6). Since the problem is symmetric with respect to $A$ and $B$, and $p$ and $(1-p)$, we can trivially derive the equation for the other two cases, which is:

$$
\frac{1}{(1-p)B}(2+\ln(\frac{(1-p)B}{pA})) \qquad (8)
$$

Thus the full function for calculating the expected number of nodes searched given $A$, $B$ and $p$ is given by the hybrid function:

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