Population based Ant Colony Optimization on FPGA

M. Guntsch[1], M. Middendorf[3], B. Scheuermann[1], O. Diessel[2], H. ElGindy[2], H. Schmeck[1], K. So[2]

[1]Institute AIFB
University of Karlsruhe (TH), Germany
{mgu,bsc,schmeck}@aifb.uni-karlsruhe.de

[2]Computer Science and Engineering
University of New South Wales, Australia
{odiessel,hossam,keiths}@cse.unsw.edu.au

[3]Institute of Computer Science
University of Leipzig, Germany
middendorf@informatik.uni-leipzig.de

Abstract

We propose to modify a type of ant algorithm called Population based Ant Colony Optimization (P-ACO) to allow implementation on an FPGA architecture. Ant algorithms are adapted from the natural behavior of ants and used to find good solutions to combinatorial optimization problems. General layout on the FPGA and algorithmic description are covered. The most notable achievements featured in this paper are a runtime reduction and including the approximation of the heuristic function by a small set of favored decisions which changes over time.

1 Introduction

The Ant Colony Optimization (ACO) metaheuristic for finding good solutions to combinatorial optimization problems (see [3] for an overview) is motivated by the behavior of real ant colonies. When ants attempt to find short paths between their nest and food sources, they communicate indirectly by using pheromone to mark the decisions they made when building their respective paths. In ACO, artificial ants search for good solutions in an iterative fashion: after m ants have constructed a solution, the best ant updates it's path in the decision graph. Afterwards, the next iteration of ants builds solutions with the updated pheromone information. This process continues until some stopping criterion is met. The pheromone information usually takes the form of a matrix, e.g. choosing j at place i in the decision graph has an associated pheromone value τ_{ij}. In Population based ACO (P-ACO), however, a population of past good solutions is maintained instead of the entire pheromone matrix (see [6]). ACO has been used to find best solutions to a number of combinatorial optimization problems, for instance in scheduling (see [8]).

Some authors have studied parallel versions of ACO algorithms. But most of these efforts use coarse-grained approaches were each processor can hold the entire problem instance and the corresponding pheromone information. So far, [7] is the only work were a more fine-grained and hardware-oriented implementation of ACO is proposed. This implementation is suitable in particular for processor arrays with a dynamically reconfigurable bus system were algorithmic tasks like bit-summation and finding the rank of a number in a set of numbers can be done very efficiently. Therefore it can not be used directly for an implementation on FPGAs.

The rest of this paper is structured as follows. In Section 2 the P-ACO approach is described. The basic mapping of our Population based ACO on the FPGA is presented in Section 3. Section 4 discusses several extensions. A conclusion is given in Section 5.

2 Population based ACO

In this section we describe in detail the type of ACO algorithm which we intend to realize on an FPGA. First we will describe the generic decision process performed by a single ant in the ACO algorithm for permutation problems. Afterwards, the Population based ACO as described in [6] is explained.

2.1 Generic Decision Process

When constructing solutions to a problem instance, ants proceed in an iterative fashion, making a number of local decisions. For a permutation problem an ant has to find a permutation π of a given set of n items so that the permutation has a minimal value with respect to a given evaluation function.

For example, for the Traveling Salesman Problem (TSP), there are n cities with pairwise distances d_{ij} between cities i and j for $i, j \in [1:n]$. The goal is to find a Hamiltonian cycle which minimizes the sum of distances covered, i.e. to find a monocyclic permutation π of $[1:n]$ which minimizes $\sum_{i=1}^{n} d_{i\pi(i)}$. An ant starts at some city and proceeds by choosing the city to visit next from its current location until the tour is complete.

For the Quadratic Assignment Problem (QAP) there are n facilities, n locations, and $n \times n$ matrices $[d_{ij}]$ and $[f_{hl}]$, where d_{ij} is the distance between locations i and j and f_{hl} is the flow between facilities h and l. The goal is to find an assignment of facilities to locations, i.e. a permutation π of $[1 : n]$ such that the sum of distance-weighted flows between facilities $\sum_{i=1}^{n} \sum_{j=1}^{n} d_{\pi(i)\pi(j)} f_{ij}$ is minimized. An ant constructs a solution by going to a randomly chosen unassigned location and placing one of the remaining facilities there, continuing to do so until no free locations/facilities are left.

The decisions an ant makes are probabilistic in nature and influenced by two factors: pheromone information τ_{ij} and heuristic information η_{ij}, each indicating how good it is to choose item j at place i of the permutation. An ant chooses according to a probability distribution over the set of valid choices S

$$\forall j \in S : \; p_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{k \in S} \tau_{ik}^{\alpha} \cdot \eta_{ik}^{\beta}} \qquad (1)$$

where parameters α and β are used to define the relative influence of pheromone values and heuristic values.

For TSP the pheromone matrix $[\tau_{ij}]$ is encoded in a city$\times$city fashion, which means that τ_{ij} is an indication of how good it was to go from city i to city j. When initializing the pheromone matrix, the diagonal elements τ_{ii} are therefore set to 0, and all other pheromone values are set to the same positive value $\tau_{init} > 0$. The heuristic information is derived from the distance between cities by setting $\eta_{ij} = 1/d_{ij}$. Typically, $\beta > 1$ is chosen since the η values tend to be very close to one another.

For QAP, the pheromone value τ_{ij} gives information about placing facility j on location i. Initially, all pheromone values are set to the same value $\tau_{init} > 0$. As a standard approach, ACO algorithms for the QAP do not make use of heuristic information (see [9]).

2.2 Solution Population

After m ants have constructed a solution, the pheromone information is updated for the ants in the next iteration. P-ACO uses a population of (previously best) solutions from which the pheromone matrix can be derived. Initially the population of solutions is empty. For the first k iterations of ants the best solution found in that iteration enters the population but no solution is removed. After that, whenever the population is updated, the best solution of the iteration enters the population and the oldest one is removed so that the size of the population remains k. This means that the population can be maintained like a FIFO-queue. Hence, when k is the size of the population each solution that enters the population has an influence on the decisions of the ants over exactly k subsequent iterations. Other possibilities for determining which solutions should enter/leave the population are discussed in [5]. The pheromone matrix is updated as follows:

- whenever a solution π enters the population, do a positive update: $\forall\, i \in [1, n] : \tau_{i\pi(i)} \mapsto \tau_{i\pi(i)} + \Delta$
- whenever a solution σ leaves the population, do a negative update: $\forall\, i \in [1, n] : \tau_{i\sigma(i)} \mapsto \tau_{i\sigma(i)} - \Delta$

Note that this results in $k + 1$ discrete possible pheromone values and is different to the standard ACO, where evaporation is used to reduce all pheromone values after an iteration (see [2, 4]).

The increment value $\Delta > 0$ and the population size k, along with the number of ants per iteration m, are all parameters of the algorithm. Recall that for initialization every pheromone value τ_{ij} is set to τ_{init} except for problems were some pheromones that do not correspond to possible choices are set to zero. Note, that the exact value of τ_{init} is arbitrary, as Δ can simply be scaled accordingly.

3 FPGA mapping of P-ACO

In the following we first give an overview of aspects to be considered when mapping ACOs into the available resources on an FPGA-chip. We then outline the algorithmic operations required by the mapping. An analysis of the area required by mapping the operations to Virtex is presented at the end of this section.

3.1 Overview

Of primary concern in mapping an ACO algorithm to fine-grained hardware such as FPGAs is the handling of the $n \times n$ arrays representing the pheromone and heuristic values. In the sequential RAM model these are stored in memory and accessed by one ant at a time. Ants are processed iteratively and after m ants the population of good ants is updated. Previous parallel implementations [7] pipe a set of ants through statically allocated $n \times n$ matrices in a systolic fashion. For hardware of fixed size, parameterizing the mapping on n limits us unduly. We have instead chosen to map a generation of m ants to the FPGA resource and process them, i.e. allow them to develop a solution over $n-1$ decision cycles, in place. As n grows our hardware requirement for processing grows linearly, not quadratically, and we envisage folding our processing hardware to fit the available area (see Section 4 for more details).

The significant challenge we then face in developing a suitable mapping is how to implement and determine the probability distribution of Equation 1. Since the membership of set S is dynamic, the usual approach is to compute

the prefix sums of the products in the numerator over the as yet unchosen elements that remain in the set at the beginning of the decision step. This suggests the need to support multiplication and addition operations, and potentially $O(n)$ time to compute the distribution, neither of which are attractive for implementation in current field-programmable technology. If we restrict ourselves to problems like QAP that do not usually consider heuristic information, the need for multiplication is avoided. Furthermore, we can observe that the population of k previous good solutions contribute at most k positive pheromone updates to the set of choices currently available to the ant. Thus, by broadcasting the k elements of the respective solutions in the population, an ant can determine how many updates apply to the current set S and which elements in S were targeted by an update. The former allows the denominator of Equation 1 to be calculated in constant time following the broadcasts. The latter can be used together with a randomly generated number to select an element remaining in S according to the probability distribution of Equation 1 in constant time. Our method relies upon buffering the addresses of elements in S whose pheromone value should be updated according to their occurrence in one of the population entries. Multiple updates to the one element give rise to multiple entries in the buffer. Scaling the number of buffered addresses by Δ and adding the size of the selection set S yields the required denominator R. Subtracting the size of S from a uniformly distributed random number $r \leq R$ then yields either the address a (index) of the element to be chosen from S directly, or its address as selected from the buffer when the difference d is scaled by $1/\Delta$. Choosing Δ to be a power of 2 further simplifies calculations by reducing the scaling operations to left and right shifts.

The significance of this approach is that the decision cycle for an ant takes $\Theta(k)$ time assuming the random number generation can be bound to $O(k)$ steps (see the next subsection for further analysis). For a problem of size n, this implies $\Theta(nk)$ time is required per ant to complete a solution. In comparison, a sequential processor will require $O(n^2)$ time per ant. The functional parallelism embodied in processing m ants in parallel on the FPGA allows m solutions to be formed in $\Theta(nk)$ time, which compares with $O(mn^2)$ time to achieve the same result on a sequential processor.

3.2 P-ACO Design on FPGA

In this section we first describe the top-level of the P-ACO algorithm. We then explain the implementation of the main modules in greater detail.

The processing flow executed by the P-ACO is presented in Figure 1. The algorithm starts by initializing the population, selection set $S := \{0, ..., n-1\}$ and problem-specific evaluation data (e.g. distance matrix d_{ij} for TSP). The pop-

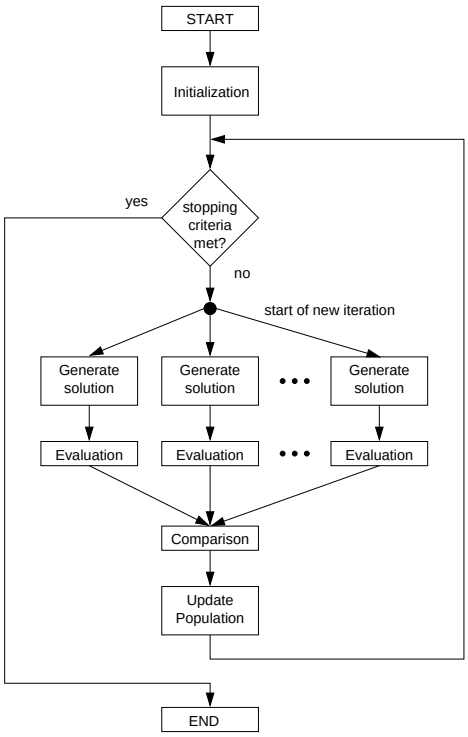

Figure 1. P-ACO processing flow

ulation of k good solutions is stored as a queue, which is initialized to an empty set $Q := \emptyset$. Afterwards m ants iteratively create solutions until some stopping criterion has been met (e.g. the maximum number of iterations has been exceeded, or the best solution has not changed over a certain number of iterations). Here m solutions are generated and evaluated in parallel. After comparing the results of m evaluations, the best of these m solutions is used to update the current population, which is accomplished by adding it to the FIFO-queue.

At high level, the mapping of the P-ACO algorithm into the corresponding FPGA design is straightforward (see Figure 2) and consists of three main modules: Population, Generator and Evaluation. Note that, for the sake of clarity, all schematic design figures only show data paths; the controlling state machines and their control signals have been omitted. The Population Module contains the queue $Q = \{\vec{q}_i \mid i = 0, \ldots, k-1\}$ of k good solutions, with $\vec{q}_i = (q_{i0}, \ldots, q_{i,n-1})$ and q_{ij} being the j-th decision in the i-th solution. The module manages all communication between the queue and the Generator Module via an $n \times m$ Crossbar. Furthermore, at the end of the current iteration it receives the best solution from the Evaluation Module, which is then inserted into the queue. The Generator Module holds m Solution Generators, one per ant. The solutions

are transferred from there to m parallel Evaluation circuits in the Evaluation Module. It is also possible to have less than m Solution Generators and Evaluation circuits, which will be discussed in greater detail in Section 4. The evaluation results of these m solutions are collected in a Comparison block, which chooses the best solution of the current iteration.

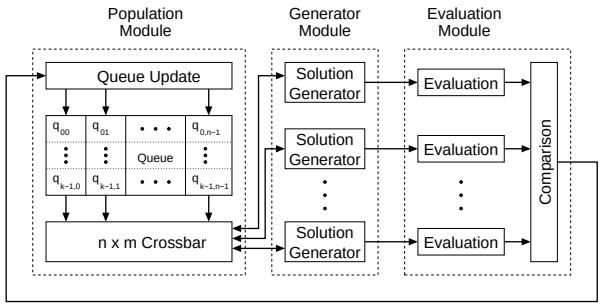

Figure 2. P-ACO design with Population, Generator and Evaluation Modules

3.2.1 Population Module

The current population of k good solutions is stored in rows $(q_{i0}, ..., q_{i,n-1}), i \in \{0, ..., k-1\}$, one row per solution, with each solution being a permutation of the numbers $0, \ldots, n-1$. The Queue is updated by inserting the update solution. If the queue is full prior to insertion, then the oldest solution is removed in order to maintain a constant queue size of k. In every iteration the Population Module receives $n-1$ requests for queue columns $(q_{0j}, ..., q_{k-1,j})$ from each Solution Generator. A request is encoded as index j of the required queue column. Since the Solutions Generators need an arbitrary and concurrent access to the queue columns, probable conflicts are avoided by an intermediate $n \times m$ Crossbar, which establishes the required data bus connections. Queue column elements $q_{ij}, j \in \{0, ..., k-1\}$ are sent sequentially via the bus connections formed.

3.2.2 Generator Module

The operating sequence of an individual Solution Generator simulates the behavior of an artificial ant constructing a solution. Each Solution Generator has its own local controlling state machine and can therefore work independently of the other Solution Generators as well as other modules on chip. The flow of instructions executed by a Solution Generator is depicted in Figure 3.

A Solution Generator starts off by filling the selection set S with the numbers $0, \ldots, n-1$ and initializing the S-Counter c, which over the course of the solution finding

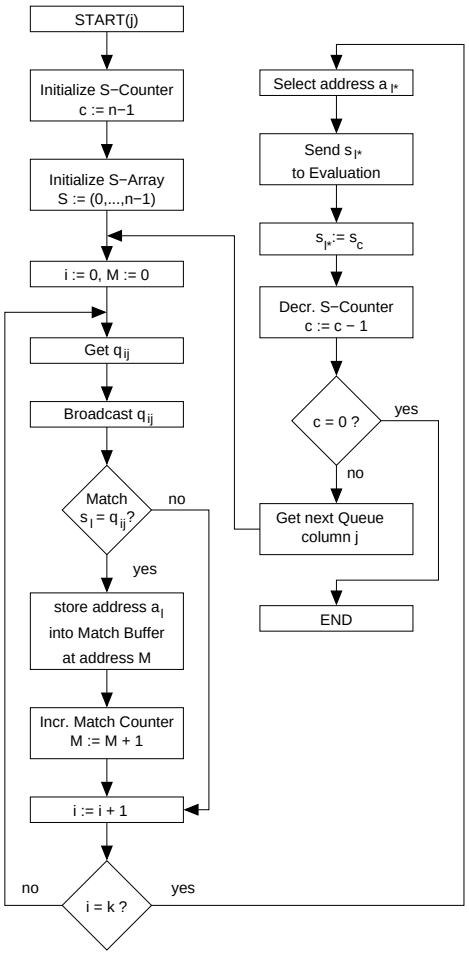

Figure 3. Solution generation

process will be the index of the last element of the S-Array, i.e. the size of S, see Figure 4. Afterwards, the first decision cycle starts with the initialization of the counters i, used for indicating the current solution of the population from

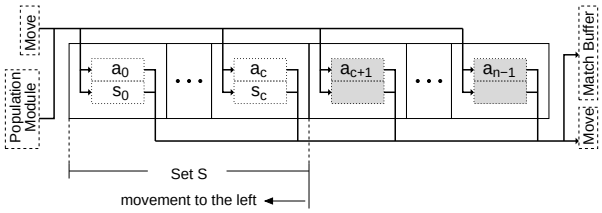

Figure 4. S-Array. $a_i = i$ is the index of cell i, and s_i is the element from S at this address. Shaded cells are deactivated, they no longer contain an element of S.

which an element is to be read, and M, which counts the number of matches between elements from the population and selection set S.

The loop which starts after initialization has the goal of writing those addresses a_l of elements s_l which match the elements received from the population from the S-Array into the Match Buffer, see Figures 5 and 6. Note that if an element matches multiple times, its address is also transferred to the Match Buffer multiple times and the M counter is increased accordingly.

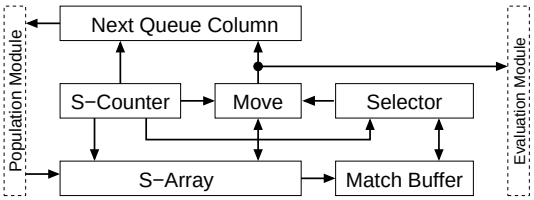

Figure 5. Solution Generator

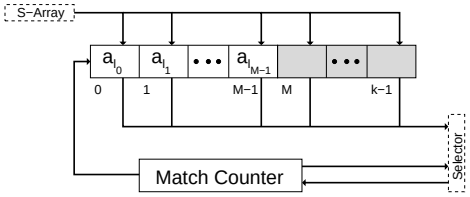

Figure 6. Match Buffer, shaded area is empty.

At the end of this loop, the values of M and c are transferred to the Selector, see Figure 7. Here, the upper bound R for the random number r is calculated and r drawn uniformly from the interval $[0, R-1]$ afterwards. Without large multiplier or divider circuits, it is difficult to draw a random number uniformly from an arbitrary interval. Hardware-based random number generators creating random bits can be used to create random numbers from the interval $[0, 2^x - 1]$ by drawing x bits. However, since R will most likely not be a power of 2, we suggest repeatedly drawing a random number $r \in [0, R'-1]$ until $r < R$, where $R' = 2^{\lceil \log(R) \rceil}$ is the smallest power of 2 greater than or equal to R. Let $p = Prob(r < R) = \frac{R}{R'}$ denote the probability of drawing a random number $r \in [0, R-1]$, where $\frac{1}{2} < p \leq 1$. Then the probability of drawing a random number $r \in [0, R-1]$ after $z-1$ unsuccessful trials is $P(Z = z) = p(1-p)^{z-1}$, decreasing exponentially in z. Hence, the expected value $E(Z)$ is

$$E(Z) = p \sum_{z=1}^{\infty} z(1-p)^{z-1} = \frac{1}{p}. \qquad (2)$$

On average, random numbers have to be drawn $E(Z)$ times, with $1 \leq E(Z) < 2$, in order to receive a uniformly distributed random number $r \in [0, R-1]$. The random bits necessary for generating the random number are created with the RNG introduced by Ackermann et al. in [1], which is well suited in terms of quality of the random bits and space requirements. Note that the individual Solution Generators can and probably will take different amounts of time to draw their respective random numbers. However, since each Solution Generator has an independent local controlling state machine, no timing problems arise from this.

Once r has been generated, we compare the number with c to determine whether an element from the S-Array or the Match-Buffer has been chosen. If $r \leq c$, then $a_{l^*} = a_r$ is the index of the selected element from S; otherwise, $a_{l^*} = a_{M^*}$, with $M^* = l_{(r-c) \gg y}$, is selected from the Match Buffer, where $\Delta = 2^y$ is the update value associated with elements in the population. Recall that the Match Buffer consists of the addresses $\{a_{l_0}, \ldots, a_{l_{M-1}}\}$. After a_{l^*} has been determined, the corresponding element s_{l^*} is read from the S-Array and sent to the Evaluation Module, which is described in Section 3.2.3. Now, all that remains is updating the set S, decrementing its counter c, and computing place j in the population from which the elements of the next cycle will be read, until $n-1$ cycles have been completed.

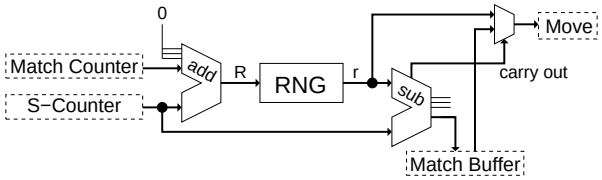

Figure 7. Selector

3.2.3 Evaluation Module

The main role of the Evaluation Module is to assign respective solution qualities to the permutations $\pi_1, \ldots, \pi_m$ which are constructed by the Solution Generators and, after having received all m solutions, to pick the best one to be sent to the Population Module as an update to the population stored in the queue. Since there are as many Evaluation Units as there are Solution Generators, there is no problem with addressing or sharing a bus for transferring data. An Evaluation Unit receives the choices made by the Solution Generator in the individual cycles immediately, and can therefore calculate the quality of the solution online, if the problem permits. For TSP for example, every time the current choice, i.e. the next city to visit, is sent to the Evaluation unit, the length of the corresponding edge is added

to the overall tour length, which is also the solution quality. In QAP, which takes considerably longer to evaluate in the unrestricted case, the newly determined products of flows and distances are added after every new placement of a facility. Most optimization problems in which an optimal permutation is sought admits some form of online quality computation.

Once the entire permutation π_i has been transferred to Evaluation Unit i, it finishes computing the corresponding solution quality and transfers both the permutation and its quality to a backup array so that Solution Generator i can immediately start building a new solution. As soon as the Comparison Unit is available, it receives the solution quality of π_i from Evaluation Unit i. If this quality is better than all previous and other currently finished π_j in the same iteration, the permutation is transferred to the Comparison Unit. Note that the Evaluation Units can deliver their solutions asynchronously since the generation of random numbers might take a different amount of time in the Solution Generators. After the Comparison Unit has determined which of the past m solutions is best, it forwards this solution π^* to the Population Module to update the population.

3.3 Area Requirements

So far the P-ACO design has been explained on a rather abstract level. In this section we present implementation ideas for some selected partial circuits together with the corresponding estimates on the required FPGA resources. We assume targeting a Virtex device of at least size XCV300.

3.3.1 S-Array cell

As described before, an S-Array consists of n cells (S-Cells). Each of them keeps two integer values, an address value a and a value s being a member of choice set S. Both values are supposed to be encoded as vectors of $N = \log n$ bits. For an S-Cell, it is desirable to store a and s in flip-flops to allow reading and writing in constant time - the alternative, which is to store in LUTs, would result in $\Theta(N)$ cycles for reading and writing. Previous observation implies that $N/2$ slices are needed to store either a or s, and a vertical arrangement facilitates a comparison with the broadcast value q.

Refer to Figure 8 for a Virtex CLB configuration that allows 2-bit values that are broadcast on horizontal wires (singles or hex) to be matched within a CLB. In this circuit 2 bits of s are stored in flip-flops. All local match signals MS_i are accumulated along the fast carry logic resources to create a wired AND. For the bottom-most CLB Cin is initialized to 1. The overall match signal MS^* is output on the $Cout$ port of the top-most CLB. MS^* is fed back, thereby spanning the complete column.

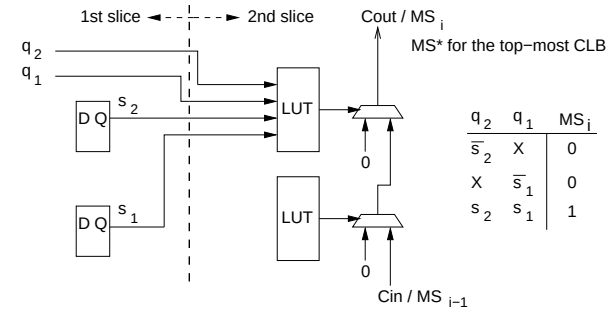

Figure 8. S-Cell circuit matching queue element broadcasts q against s-values

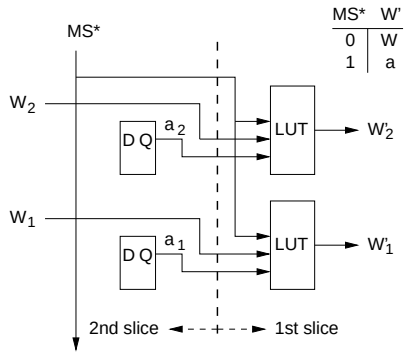

Figure 9. Circuit propagating address values a to the Match Buffer

Figure 9 depicts a Virtex CLB configuration that will propagate address value a from a flip-flop along single wires W horizontally spanning a row of CLBs. The function of

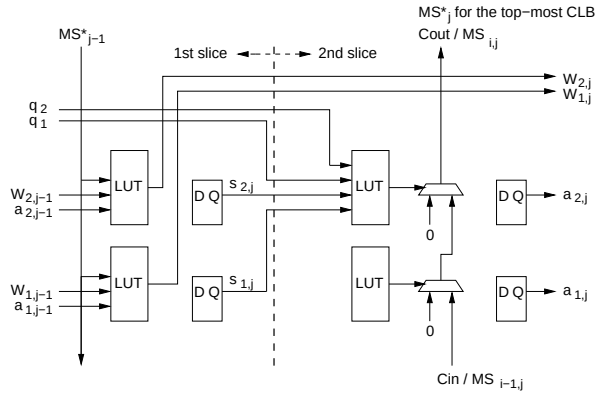

Figure 10. Top-most S-Cell circuit combining match and propagation functionality

the LUTs is to form a CREW-bus through n S-Cells. Again $N/2$ circuits per S-Cell have to be vertically aligned in a column to be able to create a bus segment of width N.

A partial S-Cell circuit is shown in Figure 10, which matches broadcast bits q with element s and forwards address bits a. We estimate 2-3 CLBs are therefore required to provide the functionality needed to support 2 bits of an a-s-pair. Since $N/2$ rows are needed for N bits, in total Nn to $1.5 \cdot Nn$ CLBs are necessary to form an S-Array.

3.3.2 Match Buffer

The Match Buffer consists of a Match Counter and memory circuitry to store up to k integer values $(a_1, ..., a_{M-1})$ of size N bits with $M \leq k$. Since in practical applications values for k are usually less than 10, the storage part can efficiently implemented by configuring N LUTs as RAM blocks $RAM_j = (R_{j,0}, ..., R_{j,15})$ with a capacity of 16 bits per block. Let $a_i = (a_{i,0}, ..., a_{i,N-1})$ with $0 \leq i \leq M-1$ be the bit representation of an integer value. We propose storing integer values in a distributed fashion: $RAM_j = (a_{0,j}, ..., a_{M-1,j})$, where $0 \leq j \leq N-1$. Since the integer values are accessed only sequentially, this method allows bit-parallel read and write operations in constant time and a total area consumption of $N/4$ CLBs. For very small k, however, matches should rather be stored in block RAM, if available. The Match Counter requires at most 1 CLB considering usual parameter settings for k (see [6]).

3.3.3 Selector

Building the Selector requires an N'-bit adder, an $N'+1$-bit random number generator and an $N'+1$-bit subtracter, where $N' = \max(\log k + y, N))$, so that in total $\Theta(N')$ CLBs are required for these structures with single wire delays.

4 Modifications

This section deals with modifications to the architectural schematic for P-ACO on an FPGA described in Section 3. Specifically, we will propose a technique for reducing the space needed by the P-ACO algorithm and discuss how to enable the algorithm to use heuristic information.

4.1 Space Constraints

Depending on the size of the FPGA designated for mapping the P-ACO algorithm, we will at some point be dealing with problem instances too large to fit the entire algorithm as introduced in Section 3. Specifically, the "height" of the P-ACO algorithm increases only logarithmically with n while the "width" increases in a linear fashion, resulting in an increasingly flat rectangle shape as a basic structure. Then, fitting the algorithm can be accomplished by folding, which is standard technique for this kind of problem.

Another method to make better use of the available space takes into account the fact that the selection set S decreases in size over time. It is possible at certain points in time to move the currently active selection sets to smaller rows. Two examples for this modification are given in Figure 11.

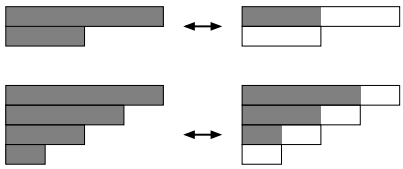

Figure 11. Example of 2- resp. 4-row configuration with decreasing size. The gray area is the part of the row which holds the set S.

The 2/4-row configurations in Figure 11 save 25/37.5% space at the cost of having the simulated ants be in different stages of completion due to their respective starting delays. The left side shows all rows after a new ant was started, the right side the moment just before the sets are shifted down and a new ant starts. Note that the shorter rows could be rearranged to make the layout more compact.

4.2 Heuristic Information

Another aspect of ant algorithms in general that was neglected in the basic layout described in Section 3 is the use of heuristic information about the problem instance. In this subsection, we describe a method of utilizing heuristic information without giving up the $O(k)$ cycle length of our FPGA implementation of the P-ACO algorithm.

The method for increasing the probability that an element be chosen is by repeating this element in a separate buffer associated with a higher weight per entry, as described in Section 3. Implementing this approach for heuristic information poses two problems: heuristic values are generally real-valued, and they exist for all elements of the set S, not just an $O(k)$ size subset.

We propose to transform a heuristic-vector $\vec{\eta} \in \mathbb{R}^n$, e.g. the distances from one city to all others in TSP, into a set of element-vectors $\{\eta'_1, \ldots, \eta'_h\}$ with $\eta'_i \in [0, n-1]^k$, where each vector η'_i holds elements which have a high heuristic value. The following method is proposed:

1) Calculate $\delta = \frac{1}{n} \sum_{i=1}^{n} \vec{\eta}_i$.

2) do the following n times

2.1) determine i so that $\vec{\eta}_i = \max_{j=1,\ldots,n} \vec{\eta}_j$

2.2) add element i to pool of numbers for building the element-vectors.

2.3) update $\vec{\eta}_i \mapsto \vec{\eta}_i - \delta$.

The quality of the approximation attainable by this method depends on the values of $\vec{\eta}_i$ and h. Deciding which elements should be placed into a given vector is accomplished by placing any one element in as many different vectors as possible and combining it with the maximum amount of other elements. The order of the elements in each vector η'_i is arbitrary. If $h > 1$, i.e. there is more than one element-vector for the given row, then we declare one of the element-vectors η'_i as active, and the other $h-1$ as inactive. Only the active element-vector affects the decision- process of the ant. After an iteration of m ants has finished, the active element-vector η'_i is replaced by some other η'_j with $j \in [1,h]\backslash\{i\}$. Note that the creation of the element-vectors $\eta'_1, \ldots, \eta'_h$ must take place before the algorithm is programed onto the FPGA. Therefore, problems which require an online computation of the heuristic values, e.g. most scheduling problems, cannot be handled by this method.

The way in which an element-vector influences the decision made by an ant is practically the same as that of the current population-vector. The respective addresses of elements which are in the element-vector as well as the selection set S are copied into a separate location called the H-Buffer, which works exactly like the Match Buffer in Figure 6. Furthermore, since the pheromone and heuristic values are multiplied by the ant algorithm, we need an additional buffer which stores the elements that are in the element-vector, the population-vector, and S. Let $\Delta_P = 2^{y_P}$ be the weight associated with the population-vector and $\Delta_H = 2^{y_H}$ the weight of the element-vector derived from the heuristic information. Then, $\Delta_{PH} = \Delta_P \cdot \Delta_H = 2^{y_P + y_H}$ is the weight for an element of which the address is stored in the PH-Buffer. Note that the combined weight of an element which is in S as well as the population- and the element-vectors is $1 + \Delta_P + \Delta_H + \Delta_{PH} = (1 + \Delta_P) \cdot (1 + \Delta_H)$, which is in accordance with Equation 1. The Selector from the basic layout in Figure 7 must be modified as well, since we can now have 4 sets of elements with different weights instead of two. These modifications, however, are essentially only doing two further subtractions and expanding the multiplexer which chooses the address to relay to the Move block in Figure 5.

5 Conclusion

We have designed a mapping of P-ACO to an FPGA architecture. In doing so, we have implemented new ways for dealing with the pheromone information which have led to significant improvements in runtime and area requirements in comparison to a sequential ant algorithm. Furthermore, we have shown possibilities for compacting the algorithm and include heuristic information in the process of constructing the solutions without asymptotically increasing runtime or required space.

Our future work will include actually implementing the algorithm on chip. Also, the effect of the discretized heuristic on solution quality will be investigated.

6 Acknowledgments

This work was supported by the Int. Office (IB/DLR) of the German Ministry of Education and Research (BMBF) within the scope of WTZ-project AUS 00/002.

References

[1] J. Ackermann, U. Tangen, B. Bödekker, J. Breyer, E. Stoll, and J. McCaskill. Parallel random number generator for inexpensive configurable hardware cells. *Computer Physics Communications*, 140(3):293–302, 2001.

[2] M. Dorigo. *Optimization, Learning and Natural Algorithms* (in Italian). PhD thesis, Dipartimento di Elettronica , Politecnico di Milano, Italy, 1992. pp. 140.

[3] M. Dorigo and G. Di Caro. The ant colony optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, 1999.

[4] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Trans. Systems, Man, and Cybernetics – Part B*, 26:29–41, 1996.

[5] M. Guntsch and M. Middendorf. Applying population based aco to dynamic optimization problems. In M. Dorigo et al., editor, *Ant Algorithms: 3rd International Workshop, ANTS2002*, volume 2463 of *Lecture Notes in Computer Science*, pages 111–122. Springer Verlag, 2002.

[6] M. Guntsch and M. Middendorf. A population based approach for ACO. In S. Cagnoni et al., editor, *Applications of Evolutionary Computing - EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN*, number 2279 in Lecture Notes in Computer Science, pages 72–81. Springer Verlag, 2002.

[7] D. Merkle and M. Middendorf. Fast ant colony optimization on runtime reconfigurable processor arrays. *Genetic Programming and Evolvable Machines*, 3(4), 2002.

[8] D. Merkle, M. Middendorf, and H. Schmeck. Ant colony optimization for resource-constrained project scheduling. *IEEE Trans. on Evolutionary Computation*, 6(4):333–346, 2002.

[9] T. Stützle and M. Dorigo. ACO algorithms for the quadratic assignment problem. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 33–50. McGraw-Hill, 1999.