Parallel Evolutionary Optimisation with Constraint Propagation *

Alvaro Ruiz-Andino[1], Lourdes Araujo[1], Jose Ruz[2], and Fernando Sáenz[2]

[1] Department of Computer Science
[2] Department of Computer Architecture
Universidad Complutense de Madrid

Abstract. This paper describes a parallel model for a distributed memory architecture of a non traditional evolutionary computation method, which integrates constraint propagation and evolution programs. This integration provides a problem-independent optimisation strategy for large scale constrained combinatorial problems over finite integer domains. We have adopted a global parallelisation approach which preserves the properties, behaviour, and theoretical studies of the sequential algorithm. Moreover, high speedup is achieved since genetic operators are coarse-grained, as they perform a search in a discrete space carrying out constraint propagation. A global parallelisation implies a single population but, as we focus on distributed memory architectures, the single virtual population is physically distributed among the processors. Selection and mating consider all the individuals in the population, but the application of genetic operators is performed in parallel. The implementation of the model has been tested on a CRAY T3E multiprocessor using two complex constrained optimisation problems. Experiments have proved the efficiency of this approach since linear speedups have been obtained.

1 Introduction

In this paper we present the parallelisation, for a distributed memory architecture, of an integration of evolution programs and constraint propagation techniques. Evolution programs and constraint propagation complement each other to efficiently solve large scale constrained optimisation problems over finite integer domains [5, 6]. Evolution programs [3] are an adequate optimisation technique for large search spaces, but they do not offer a problem-independent way to handle constraints. Constraint propagation and consistency algorithms, which prune the search space before and while searching, are valid for any discrete combinatorial problem.

Evolutionary programming is inherently parallel. Moreover, in this case, task granularity is increased because of coarse-grained genetic operators, since they perform a search embedding constraint propagation.

There are three main approaches to parallelise an evolution program [2, 1]: global parallelisation, island model, and hybrid algorithms. We have adopted a global parallelisation approach because of the following reasons:

* Supported by project TIC95-0433.

A.E. Eiben et al. (Eds.): PPSN V, LNCS 1498, pp. 270-279, 1998.

- Properties, behaviour, and theoretical studies of the sequential algorithm are preserved.
- Crossover is coarse-grained, as it implies searching in a discrete search space performing constraint propagation.
- The higher communications rate of a global parallelisation versus other approaches does not significantly penalises speedup, since modern distributed memory multiprocessor provide fast, low-latency asynchronous read/write access to remote processors' memory, avoiding rendevouz overhead.

Our approach is based on a single virtual population physically distributed among the processors, in such a way that each processor owns a partial copy of the population. Selection of chromosomes to be replaced, to be mutated, and parents to take part in crossover, is performed in a centralised manner by a distinguished processor (*master*), but the application of genetic operators is performed in parallel. Coordination is achieved through annotations in mutual exclusion on the local memory of the master processor. Scheduling of pending operations is performed in a dynamic self-guided way, following a set of rules to minimise the number of chromosomes to fetch from remote processors.

The rest of the paper is organised as follows. Section 2 summarises the main points of the combination of constraint propagation and evolution programs presented in [6]. Section 3 presents the parallelisation model, the parallel algorithm, and the work scheduling policy. Section 4 describes the results obtained for two complex constrained optimisation problems. Finally Section 5 discusses the conclusions.

2 Constraint Propagation and Evolution Programs

Many complex search problems such as resource allocation, scheduling and hardware design [9] can be stated as constrained optimisation problems over finite integer domains. Constraint programming languages are based on a constraint propagation solver embedding an Arc-Consistency algorithm [4, 8], an efficient and general technique to solve finite domain constraint satisfaction problems. Arc-Consistency algorithms eliminate inconsistent values from the domains of the variables, reducing the size of the search space both before and while searching.

Constraint propagation techniques and evolution programs complement each other. Constraint solving techniques opens a flexible and efficient way to handle constraints in evolution programs, while evolution programs allow searching for solutions in large scale constrained optimisation problems. Other approaches to handle constraints in evolution programs require to define problem specific genetic operators, whereas constraint propagation embedded in evolution program results in a problem-independent optimisation strategy for constrained optimisation problems. This section summarises the main points of the integration.

Constraint Propagation. A constraint optimisation problem over finite integer domains may be stated as follows. Given a tuple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C}, f \rangle$, where $\mathcal{V} \equiv \{v_1, \cdots, v_n\}$, is a set of domain variables; $\mathcal{D} \equiv \{d_1, \cdots, d_n\}$, is the set of an initial

finite integer domain (finite set of integers) for each variable; $\mathcal{C} \equiv \{c_1, \cdots, c_m\}$, is a set of constraints among the variables in $\mathcal{V}$, and $f : N \times \cdots \times N \rightarrow N$ is the objective function to optimise. A constraint $c \equiv (V_c, R_c)$ is defined by a subset of variables $V_c \subseteq \mathcal{V}$, and a subset of allowed tuples of values $R_c \subseteq \bigotimes_{i \in \{j / v_j \in V_c\}} d_i$, where $\bigotimes$ denotes Cartesian product. The goal is to find an assignment for each variable $v_i \in \mathcal{V}$ of a value from each $d_i \in \mathcal{D}$ which satisfies every constraint $c_i \in \mathcal{C}$, and optimises the objective function f. A constraint $c \in \mathcal{C}$ relating variables v_i and $v_j \in \mathcal{V}$, is *arc-consistent* with respect to domains d_i, d_j iff for all $a \in d_i$ there exists $b \in d_j$ such that (a, b) satisfies the constraint c, and for all $b \in d_j$ there exists $a \in d_i$ such that (a, b) satisfies the constraint c. A constraint satisfaction problem is arc-consistent iff all $c_i \in \mathcal{C}$ are arc-consistent with respect to $\mathcal{D}$. An Arc-Consistency algorithm takes as input arguments a set of constraints to be satisfied and an initial finite integer domain for each variable. The algorithm either detects inconsistency (a variable was pruned to an empty domain), or prunes the domain of each variable in such a way that arc-consistency is achieved.

Integration of evolution programs and constraint propagation implies coming up with a solution for chromosome representation, chromosome evaluation, and genetic operators' design.

Chromosome Representation. In order to take advantage of the arc-consistency techniques embedded in the constraint solver, a chromosome is an array of finite integer domains, that is, a sub-space of the initial search space. Moreover, chromosomes are arc-consistent solutions (AC-solutions) generated by means of genetic operators based on an Arc-Consistency algorithm.

Chromosome Evaluation. Searching for a solution using an Arc-Consistency algorithm involves evaluating the AC-solution generated, since the objective function is assimilated to a domain variable whose domain is pruned while searching. The objective function f is defined by means of constraints, leading to an extended function $f' : N^* \times \cdots \times N^* \rightarrow N^*$. f' takes as arguments finite integer domains and returns a finite integer domain. A dual evaluation is used: a *fitness* value is computed from the pruned domain of the variable to optimise, but, as we are dealing with non fully determined solutions, a *feasible* value is used to take into account the probability that a feasible solution lies within the AC-solution.

Genetic operators implement stochastic heuristic arc-consistent searches, taking previous AC-solutions as an input information to guide the search for a new AC-solution. Arc-consistency is achieved at each node of the search space, removing inconsistent values from the domain of the variables. Figure 1 shows the `AC-crossover` function. Given two AC-solutions, the AC-crossover operator generates a new AC-solution. K (a random value between 1 and n) randomly chosen variables, $v_{perm[1]}$ through $v_{perm[K]}$, are constrained to domains from first parent. Then, Arc-Consistency algorithm is invoked, pruning the search space. Remaining variables, $v_{perm[K+1]}$ through $v_{perm[n]}$, are constrained one by one to domains from second parent, performing Arc-Consistency at each step. If inconsistency is detected at any step, variable's domain is left unchanged.

```
function AC-crossover( AC-Sol1, AC-Sol2 ) : AC-solution;
begin
   perm := random-permutation(1, n);
   K := random-int-between(1, n);
   for i = 1 to K do
      v_perm[i] := AC-Sol1[perm[i]];
   end-for;
   Perform Arc-Consistency, pruning search space
   for i = K+1 to n do
      v_perm[i] := AC-Sol2[perm[i]];
      Perform Arc-Consistency, pruning search space
      if Inconsistent then undo assigment;
   end-for;
   return generated AC-Solution
end;
```

Fig. 1. AC-Crossover

3 Parallelisation Model

A global parallelisation of the presented constrained optimisation evolution program is expected to achieve high speedups, since the constraint propagation leads to coarse-grained genetic operators. Global parallelisation implies a centralised population. Shared memory architectures support an straight implementation of this approach, whereas distributed memory architectures may suffer from communications overhead. We propose a global parallelisation model for a distributed memory architecture based on a virtual centralised population, physically distributed among the processors in order to reduce communications. Target architecture is any modern distributed memory multiprocessor that allows fast asynchronous read/write access to remote processors' memory. This feature places them in a middle point between traditional shared and distributed memory architectures.

The data distribution model, appearing in Figure 2, can be summarised as follows:

- The population is distributed among the processors in the system. Each processor owns a subset of the population and a local *localisation table* indicating a processor where non-local chromosomes can be found.
- One processor of the parallel system is distinguished as *master*. This processor behaves as any other, but it is also in charge of the sequential part of the algorithm, and keeps the shared *operation table*.
- The *master* produces the *operation table*, which reflects chromosomes selected to survive, to be mutated, and to take part in crossover (global mating). The operation table is broadcasted at the beginning of every generation, so each processor has a local copy of it.
- Genetic operations are performed in parallel. Coordination is achieved by means of atomic test&swap on the master processor's operation table. A

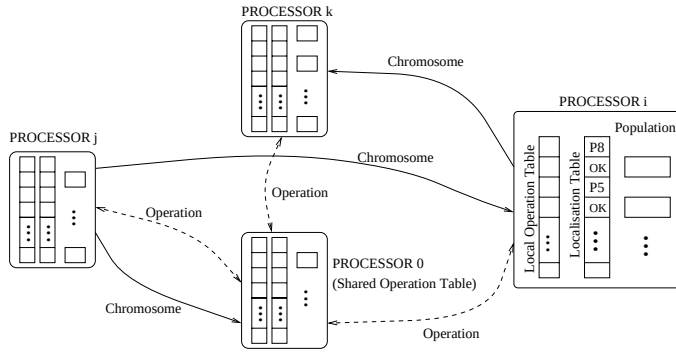

Fig. 2. Data distribution model. Solid arrows represent fetching a remote chromosome. Dotted arrows represent mutual exclusion access to the shared operation table.

processor may need to fetch (asynchronously) a chromosome from a remote processor's memory in order to perform the selected genetic operation.

At the beginning of each generation, each processor owns a subset of the population formed by:

- chromosomes generated by itself in the previous generation.
- chromosomes from the previous population fetched from a remote processor but not replaced in the current population (steady-state approach). Therefore, a chromosome may be present at many processors.

Figure 3 shows the algorithm executed in each processor. Initially a subset of the population is generated (line 1). Every time a new chromosome is generated, its evaluation (fitness and feasible values) are asynchronously written to the master's memory. Lines 2 to 14 enclose the main loop; each iteration produces a new generation. Synchronisation is needed at the beginning of each generation (line 3), in order to perform the global mating. The master establishes the genetic operations to generate the next population (line 5), filling the operation table, which is broadcasted to every processor (line 7). The loop in lines 8 to 12 performs genetic operations (crossover or mutation) until there are no more left. A processor may perform any of the pending operations (line 10), so it may need to fetch chromosomes from a remote processors' memory (line 9). The resulting offspring is kept in local memory, but the evaluation values are asynchronously written to master's memory (line 11).

Scheduling of pending operations is performed in a dynamic self-guided way, following a set of rules to minimise the number of chromosomes to be fetched from remote processors. Function `Fetch_Operation()` (line 8), consults the local copy of the operation table and the localisation table, choosing an operation to perform. In order to minimise the need to fetch remote chromosomes, the local operation table is scanned selecting operations in the following order:

```
    Procedure Parallel_AC-Evolution;
    begin
 1    Generate a subset of the initial population;
 2    while not termination() do
 3       Synchronisation;
 4       if I-am-the-Master then
 5          Generate-Matings-and-Mutations(Operation_Table);
 6       end-if;
 7       Broadcasting of Operation_Table;
 8       while Fetch_Operation(Operation_Table, Localisation_Table) do
 9          Fetch parents, if necessary, updating Localisation_Table;
10          Perform AC-Crossover (or AC-Mutation);
11          Write fitness-feasible to Master;
12       end-while;
13       Update(Localisation_Table);
14    end-while;
    end;
```

Fig. 3. Parallel constrained evolution program.

- Crossover of two local chromosomes.
- Mutation of a local chromosome. or crossover of a local chromosome with a remote one.
- Mutation or crossover of remote chromosomes.

Once an operation is selected, the corresponding entry of the shared operation table is tested and updated in mutual exclusion. If the selected operation has already been performed by another processor, the local operation table is updated, and another operation is chosen. Otherwise, the processor writes its unique processor number in the shared operation table. Once every operation has been performed, local copies of the operation table reflect which processor has generated the new chromosomes, allowing to properly update the localisation table (line 13) for the next generation, discarding local copies of outdated chromosomes.

Figure 4 shows an example of operation fetching. Processor 1 (P1) selects in the first place the crossover operation (XV) that will replace chromosome number 3, because both parents (chromosomes 3 and 8) are in its local memory. P1 successfully test and writes its processor number in the shared operation table. P2 behaves similarly with respect to operation 2. Once P1 has finished the crossover operation, it proceeds to select operation 1, as it owns one of the involved parents, writing its number in the shared operation table. P2 also tries to fetch operation 1, but it finds that operation has been already selected by processor 1, so P2 updates its local operation tale and proceeds to select a new operation.

Figure 5 shows the time diagram for a generation. There is a sequential time fraction due to the generation and broadcast of the operation table (T_s). The

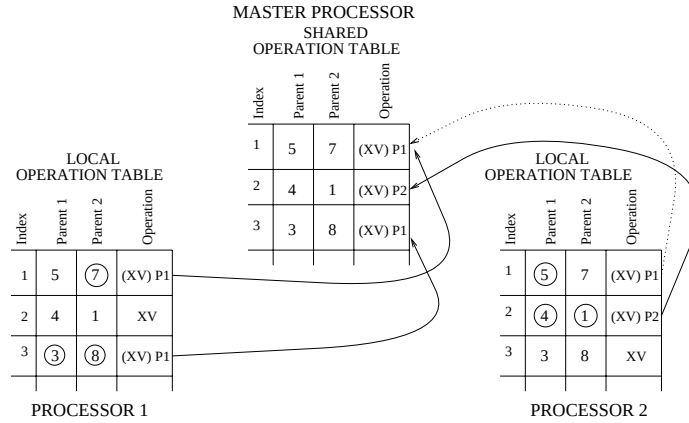

Fig. 4. Fetch Operation example. Circled numbers denote local chromosomes. Abbreviation (genetic operation to perform) in brackets denotes initial value. Solid arrows denote successful test&write operations. Dotted arrows denote unsuccessful test&write operations on the shared operation table.

time to perform a crossover in parallel (T_{xvp}) is the sequential time (T_{xvs}), increased with the time to select an operation (T_{fo}), the time to fetch the parents (T_{fp}) (only if necessary), and the time to write the evaluation values in master's memory (T_{we}). Policy to select operations favours choosing operations among local chromosomes, therefore it is expected to frequently avoid the overhead due to T_{fp}. Since genetic operators are search procedures, they can have a significantly different grain. The dynamic self-guided scheduling of the algorithm balances work load, thus reducing idle time T_{id}, introduced by the necessary synchronisation between generations.

Linear speedups will be obtained if communications overhead —T_{fo}, T_{we} and T_{fp}— is much smaller that genetic operation granularity (T_{xvs}), and when the number of genetic operations per generation is much greater than the number of processors. In this situation T_{id} and T_s are much smaller than T_p.

4 Experiments

Our system CSOS (Constrained Stochastic Optimisation System) implements the presented parallelisation model. Experiments have been carried out on a CRAY T3E multiprocessor, a distributed memory parallel system with a low latency and sustained bandwidth. Processing elements in the T3E are connected by a bi-directional 3-D torus network achieving communication rates of 480 Mbytes/s. Parallel programming capabilities are extended through the "Cray Shared Memory Library" and MPI2, which allows fast asynchronous read/write access to remote processors' memory.

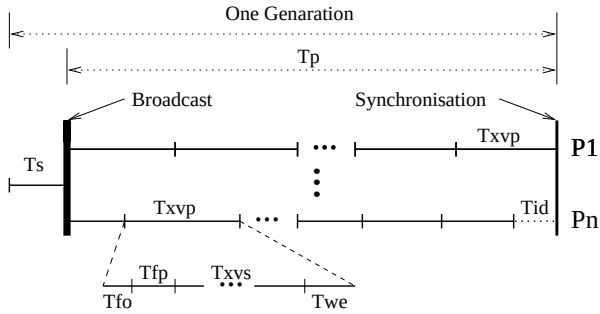

Fig. 5. Time diagram for a generation. T_s = sequential fraction (global selection for mating). T_p = parallel execution of genetic operators. T_{xvp} = parallel genetic operation. T_{xvs} = sequential genetic operation. T_{fo} = fetch an operation from master. T_{fp} = fetch remote parents. T_{we} = write evaluation values to master. T_{id} = waiting for synchronisation.

CSOS has been used to solve a number of real life constraint optimisation problems. Results obtained in two of them are reported: a VLSI signal channel routing problem, and a devices-to-FPGA mapping problem.

Channel routing consists in assigning a pair layer/track to each net from a given set of connections such that no routing segments overlap each other. The objective function to be minimised is the sum of the lengths of routing paths for a given number of tracks per layer. A chromosome is 432 words long, and average time to perform a crossover is 35 ms. Sequential execution time is 23 minutes.

A multi-FPGA (Field Programmable Gate Array) is a programmable integrated circuit that can be programmed using the hardware description language VHDL. The goal is to map the VHDL program process network into a given multi-FPGA, observing size, adjacency, and capacity constraints, minimising the maximum occupation. A chromosome is 180 words long, and average time to perform a crossover is 300 ms. Sequential execution time is 60 minutes.

Problem formulation has been programmed with a Constraint Logic Programming language over finite integer domains [7]. Therefore, problem constraints, input data, and technology-dependent parameters can be flexibly modified.

Experiments have investigated the influence of the parameters of the genetic algorithm on the speedup. A particular issue of the model affecting the speedup –the ratio of chromosomes fetched from a remote processor– is also reported. Each reported result is the average of ten executions with a different random seed.

Figure 6 shows speedup obtained for different parameter settings, all of them leading to the same number of genetic operations. The speedup is almost linear in all cases. This means that times due to communications overhead (T_{fo}, T_{fp} and T_{we}), described in Figure 5 are negligible in comparison with time to perform

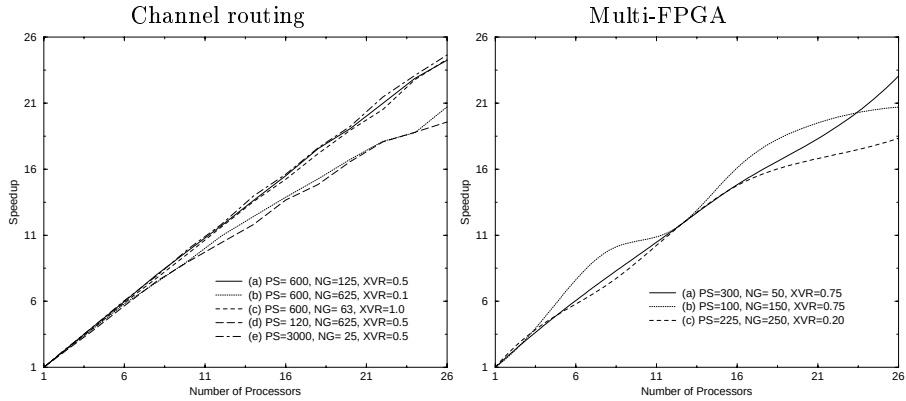

Fig. 6. Speedup obtained for different parameter settings. PS = Population size. NG = No. of generations. XVR = Ratio of population replaced with offsprings.

a crossover T_{xvs}. A lower number of crossovers per generation (small population and/or low crossover ratio) implies a higher sequential fraction and a higher idle time thus reducing speedup. Therefore, the settings scaling better, (a), (c) and (e) for the Channel routing problem and (a) for the FPGA's problem, are those with both higher population size and ratio of population replaced.

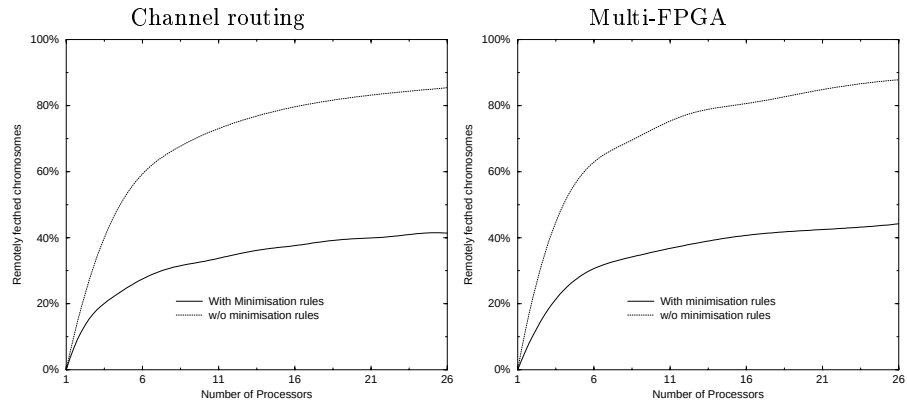

Fig. 7. Percentage of chromosomes fetched from a remote processor, with and without the minimisation policy.

Figure 7 illustrates the efficiency of the policy for selecting genetic operations, displaying the percentage of chromosomes that had to be fetched from a remote processor versus the number of processors. Solid line corresponds to the self-guided scheduling using the minimisation rules described in Section 3. Dotted

line corresponds to select the first pending genetic operation. Minimisation rules halves the percentage of chromosomes fetched from a remote processor.

5 Conclusions

We have developed a parallel execution model for a non-traditional evolutionary computation method, which integrates constraint propagation techniques and evolution programs. The system, aimed to solve constrained combinatorial optimisation problems over finite integer domains, is appropriate for parallelisation due to the coarse-grain of genetic operations. The model is devoted to run on modern distributed memory platforms, which provide communications times close to those of shared memory architectures. This characteristic justifies working with a single virtual population, though distributed across the system.

The parallel version of our system CSOS, which implements the proposed model, has been ported to a CRAY T3E, a distributed memory parallel multiprocessor. Two complex constrained optimisation problems over finite integer domains, coming from the field of hardware design, have been used to test the efficiency of the parallel model. Linear speedups have been obtained when increasing the number of processors, thus proving our hypothesis that communication overhead is negligible versus genetic operation execution times. Measurements have been taken in order to check the effectiveness of genetic operations scheduling policy. Results reveal that the percentage of chromosomes fetched from remote processors diminishes by half using our policy.

References

1. Cantú-Paz, E. A survey of parallel genetic algorithms. IlliGAL Report No. 97003 (1997).
2. Grefenstette, J.J. Parallel adaptive algorithms for function optimisation. Tech. Rep. No. CS-81-19. Nashville, TN: Vanderbilt University Computer Science Department. (1981)
3. Michalewicz, Z.: Genetic algorithms + Data Structures = Evolution Programs. 2nd Edition, Springer-Verlag (1994).
4. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence 28 (1996) 225-233.
5. Paredis, J.: Genetic State-Search for constrained Optimisation Problems. 13th Int. Joint Conf. on Artificial Intelligence (1993).
6. Ruiz-Andino A., Ruz, J.J. Integration of Constraint Programming and Evolution Programs: Application to Channel Routing. 11th Int. Conf. on Industrial Applications of Artificial Intelligence. LNAI 1415, Springer-Verlag (1998) 448-459.
7. Ruiz-Andino A. CSOS User's manual. Tech. Report No. 73.98. Department of Computer Science. Universidad Complutense de Madrid, (1998).
8. Van Hentenryck P., Deville, Y., Teng C.M.: A generic Arc-consistency Algorithm and its Specialisations. Artificial Intelligence 57 (1992) 291-321.
9. Wallace, M.: Constraints in Planing, Scheduling and Placement Problems. Constraint Programming, Springer-Verlag (1994).