International Journal of Computational Intelligence and Applications
Vol. 4, No. 4 (2004) 375–400
© Imperial College Press

A PARALLEL GENETIC ALGORITHM FOR THE GEOMETRICALLY CONSTRAINED SITE LAYOUT PROBLEM WITH UNEQUAL-SIZE FACILITIES

HAIDAR M. HARMANANI, PIERRETTE P. ZOUEIN and AOUNI M. HAJAR
Department of Computer Science, Lebanese American University
P. O. Box 36, Byblos, 1401 2010, Lebanon

Received 7 May 2003
Revised 17 February 2004

Parallel genetic algorithms techniques have been used in a variety of computer engineering and science areas. This paper presents a *parallel genetic algorithm* to solve the site layout problem with unequal-size and constrained facilities. The problem involves coordinating the use of limited space to accommodate temporary facilities subject to geometric constraints. The problem is characterised by affinity weights used to model transportation costs between facilities, and by geometric constraints between relative positions of facilities on site. The algorithm is parallelised based on a message passing SPMD architecture using parallel search and chromosomes migration. The algorithm is tested on a variety of layout problems to illustrate its performance. In specific, in the case of: (1) loosely versus tightly constrained layouts with equal levels of interaction between facilities, (2) loosely versus tightly packed layouts with variable levels of interactions between facilities, and (3) loosely versus tightly constrained layouts. Favorable results are reported.

Keywords: Facility layout problem; site layout problem; genetic algorithms; parallel computing.

1. Introduction

The layout problem is an NP-complete combinatorial optimisation problem[1,2] that has applications in many contexts including construction site planning, architectural space planning, manufacturing systems, bin packing, traffic minimisation, and VLSI design. In the area of production and manufacturing systems, the facility layout problem (FLP) was the most treated in the literature and was originally formulated in 1963 by Armour and Buffa.[3]

The layout problem was formulated differently depending on the application domain. Differences stem from the assumptions made on the shape, area, or dimensions of the objects to be positioned; the relationships that govern their relative positioning, and the constraints on their relative temporal and spatial positions. The objective is to optimise the total benefit, production cost, material handling cost or traffic flow. For example, in the VLSI chip layout problem the objective is to minimise the chip layout area. The area is influenced by the routing space which is the area between the cells occupied by the signal net wirings. Thus, the routing task is usually integrated with the placement task. Some of the early classical work on VLSI layout and floor-planning was done by Cohoon *et al.*[4–6]

The site layout problem consists of allocating temporary facilities and other resources like materials, equipment and crew workspaces needed by construction activities on a construction site. The objective is to minimise material handling costs modeled as the sum of the weighted distance between pairs of facilities.

The facility layout problem consists of determining "good" locations of a set of departments of manufacturing cells on a planar site. The objectives described by the word "good" are not all quantitative in nature and are not easily translatable into quantitatively measurable criteria. One objective is to minimise material handling cost. Yet, manpower requirements, work-in-process inventory, flow of information play an important role as well. In all cases, the prevailing assumption is that the proximity between certain departments is more favorable than others. The aim is thus to arrange the departments in such a way that the desired proximity is satisfied.

The layout problem considered in this paper is characterised by rectangular facilities with fixed dimensions and not just an aspect ratio to control their shape where both the length and width of the departments are given. Furthermore, constraints on their relative positions are not only governed by non-overlap constraints. Geometric constraints such as minimum distance, zoning, and orientation constraints can be defined to restrict their relative positions. This paper presents a parallel genetic algorithm for solving this class of layout problems.

1.1. *Related work*

By definition, the facility layout problem is a simple assignment of N facilities to N locations on the shop floor. If there are no special restrictions on the locations of specific facilities or relative location of a subset of facilities, each permutation of N locations is a feasible solution to the layout problem.

For several decades, there has been research on this subject. Kusiak *et al.*,[7] and Meller *et al.*[8] presented a detailed review of the different formulations of the FLP problem and the variety of algorithms. One can roughly distinguish between two types of problem formulations: (1) assigning a finite number of departments N to a finite number of predetermined sites M with $N \leq M$, and where the largest department can fit into the smallest site or (2) placing departments with a given area on the shop floor by subdividing the available floor area successively and placing departments so as to minimise some distance-based cost function and non-overlap. The first formulation yields a Quadratic Assignment Problem (QAP) formulation while the second formulation yields a nonlinear programming or mixed integer programming (MIP) formulation. Such mathematical programming techniques used to solve the traditional layout problem such as in Kouvelis *et al.*,[9] Ho *et al.*,[10] Bozer *et al.*,[11] Van Camp *et al.*,[12] Gilmore,[13] and Lawler,[14] have led to

NP-complete combinatorial optimisation problems that share the difficulty that the computational complexity grows rapidly with the number of departments. Hence many suggested algorithms for treating these problems try to find heuristically good solutions instead of aiming at global optimality of the solution. Heuristic approaches include improvement methods based on 2-or 3-way of facilities as in Armour and Buffa[3] or graph theoretic approaches such as Feng *et al.*[15] and Kouvelis *et al.*[9] Other approaches[12,16] used multistage techniques where rectangular facilities are first approximated by circles with a slightly larger area than an area of the actual facilities. An optimal arrangement according to the flow between the centroids of the circles is computed, and the exact shapes of the facilities are determined in a second step after fixing their positions.

But because of the computational cost and solution quality, researchers have turned away from mathematical programming based techniques to artificial intelligence based techniques[17–19] and random search heuristics such as simulated annealing[20,21] and neural networks.[23–25] For example, Tsuchiya *et al.*[22] used a $N \times N$ neurons with the objective to locate N facilities on the N-square network. Tam[21,26] used a tree representation for the facility layout problem where facilities are initially clustered according to their connectivity. The clusters characterise subtrees in a tree containing all facilities. Each inner node of the tree is labeled with the arrangement of the patterns represented by its subtrees. The structure of the tree and the mapping of the facilities to its leaves is fixed during the first step, then an optimal labeling of all inner nodes is computed using a simulated annealing[26] or a genetic algorithm.[21]

Genetic algorithms were developed namely for the QAP formulations of the FLP and for the class of FLP with unequal area facilities where constraints on permitted department shapes are specified by a maximum allowable aspect ratio for each department. In the case of the latter, Tate *et al.*[27,28] used a bay structure to model solutions where the floor area is sliced into bays and departments are assigned to bays. Each solution is modeled by two chromosomes: the first represents the sequence of departments, bay by bay and the second contains an encoding of the number of bays and the break points where they occur. Kado *et al.*[29] compare different implementations of genetic algorithms based on Tam's representation, hybridised with clustering methods. They extend Tam's work by searching the space of all possible trees while Garces-Perez *et al.*[25] use a tree representation without clustering and solve the FLP using a genetic programming approach. Although not rooted in mathematical programming, these random search techniques have proved to be powerful techniques to solve many combinatorial problems including the layout problem rather efficiently.

For the site layout problem, formulations ranged from a simple allocation of N facilities to M predefined sites where the largest facility can fit in the smallest site[30,31] to more complex formulations where facilities have different and well-defined size and dimensions and have in addition to proximity preferences between them, hard constraints on their relative locations such as minimum and maximum distance constraints, zoning constraints.[32–34] While the first class of site layout problems reduces to a QAP formulation, the second class leads to a non-linear or MIP formulation. Since both formulations are NP-complete, heuristic or suboptimal methods[34] were investigated using expert systems,[32] simulated annealing[30] and only recently using evolutionary approaches. Genetic algorithms were developed mainly for the first class of problems such as in Li and Love,[31] and Cheung *et al.*[35] where the problem was modeled as a simple assignment problem. To our knowledge, except for the work done by Harmanani *et al.*,[36] and Zouein *et al.*,[37] no GAs were developed for the second class of site layout problems which is the subject of this paper. The problem considered in this paper is also different from the various versions of FLPs treated in the literature and solved using parallel GAs.

The remainder of this paper is organised as follows. Section 2 gives a brief overview of genetic algorithm while Sec. 3 describes the constrained site layout problem. Section 4 describes the parallel formulation for the constrained site layout problem considered in this paper. Results are presented in Sec. 5. The paper concludes in Sec. 6 with a discussion of the capabilities and limitations of the proposed approach.

2. Genetic Algorithms

Genetic algorithms[38,39] (GA) is a stochastic combinatorial optimisation technique that mimics some principles of natural evolution in order to solve optimisation problems of high complexity. A group of randomly initialised points of the search space (individuals) is used to search the problem space. Each individual encodes all necessary problem parameters (genes) as bit strings, vectors, or graphs. Each gene represents one of the parameters to be optimised. The population evolves following a parody of Darwinian principle of the survival of the fittest. Individuals are selected according to their quality, measured by a fitness function, to produce offsprings and to propagate their genetic material into the next generation. Each offspring undergoes a sequence of operators (such as mutation, inversion, crossover) with a certain probability to provide diversity of the population avoiding premature convergence to a single local optimum. The iterative process of selection and combination of "good" individuals should yield even better ones, until a solution is found or a certain stop criterion is met. The main advantage of using a genetic algorithm is that it only needs an objective function with no specific knowledge about the problem space. The challenge, however, remains in finding an appropriate problem representation that results in an efficient and successful implementation of the algorithm.

Sequential GAs have been successful in various domains. However, Parallel Genetic Algorithm (PGA) provide improvements over sequential GA when they get trapped in sub-optimal region of the search space thus becoming unable to find better solutions. Furthermore, PGAs are useful when the population needs to be large and the fitness evaluation is CPU time consuming. However, the most important advantage of PGAs is that in many cases they provide better performance than a single population-based algorithm. The reason is that multiple populations permit speciation, a process by which different populations evolve in different directions toward different optima.[40]

During optimisation, the population size is very critical; if it is too small, the genetic algorithm could converge prematurely without effectively scanning the feasible region for the other possible solutions. On the other hand, a population size that is too large would be highly costly in terms of processing time and power. Important factors in determining the population size are the number of parameters involved in the optimisation process and the chromosome length. The chromosome length depends not only on the number of parameters, but also on the precision and the range of values that need to be covered for each of these parameters. This establishes a correlation between the searchable space, the chromosome length and the population size. It has been shown that in order to improve the performance of the GA without limiting its capabilities, a dynamic and parallel GA is needed.

3. The Geometrically-Constrained Site Layout Problem

3.1. *Problem description*

The layout problem, modeled in this paper, is characterised by rectangular layout objects with fixed dimensions representing the facilities to be positioned on site. Facilities can be positioned in one of two orientations only: 0° or 90° orientation. In addition, facilities can have 2-dimensional constraints on their relative positions: namely minimum and maximum distance, orientation, and non-overlap constraints. Minimum and maximum distance constraints limit the distance between the facing sides of two facilities in the X or Y direction to be greater than or less than a predefined value respectively. Distance constraints can be used to model equipment reach or general clearance requirements. Orientation constraints limit a facility's position to be to the *North*, *South*, *East*, or *West* of another reference facility. These constraints can be used to locate access roads or gates with respect to the main facility. Non-overlap constraints are the default constraints that restrict the positions of any two facilities from overlapping. The objective is to find a feasible arrangement for all layout objects within the site space that minimises the sum of the weighted distances separating the layout objects.

3.2. *Chromosomal encoding*

A population is a collection of chromosomes that evolve in order to yield an optimum or a sub-optimum solution. A chromosome represents a layout solution and it is coded as a vector whose length is equal to the number of site facilities, n. Each facility R_i is represented by the coordinates of its position on site, $\langle X_i, Y_i \rangle$ and its dimensions, $\langle L_i, W_i \rangle$. In order to facilitate the insertion and update of facilities on the site, a facility maintains references to the facilities that surrounds it in the four

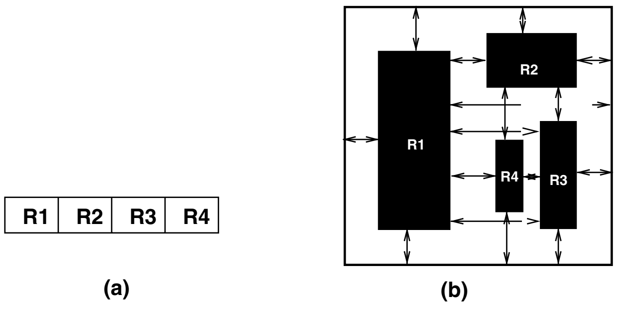

Fig. 1. (a) Chromosome representation, (b) corresponding site model.

directions. Furthermore, there is a reference from the site to the nearest block. The references are used to check for facilities overlap. A sample chromosome with four facilities is shown in Fig. 1.

3.3. *Cost function*

The objective of the facility layout problem described in this paper is to find a feasible arrangement for all layout objects within the site space that minimises the sum of the weighted distances separating the layout objects. Thus, the objective or fitness function to be optimised is:

$$Z = \sum \sum_{i<j} (w_{ij} \times d_{ij}). \tag{1}$$

Where w_{ij} is the affinity weight between objects i and j which could be used to represent the flow or the unit transportation cost between i and j, and d_{ij} is the rectilinear distance separating objects i and j. A feasible arrangement is obtained by finding positions for all layout objects that satisfy the 2-dimensional constraints between them.

3.4. *Initial population*

Chromosomes in the initial population are generated randomly by applying a sequence of mutation operations. First, blocks with user-defined fixed positions on site are added to a chromosome; these blocks cannot be moved by the algorithm. Next, mutation is applied on the remaining blocks by randomly selecting one block at a time. If applying mutation on a given block results in an unfeasible insertion coordinates, then mutation is repeated on the same block up to 100 times to find a feasible position for that block. If after 100 trials no feasible position is found, this block is left out of the chromosome. Hence, this method may result in chromosomes representing partial layout solutions. The population is repaired in this case by adding the missing blocks using a special operator (*AddMissingBlocks*).

3.5. *Reproduction*

Reproduction is the artificial version of natural selection, a Darwinian survival of the fittest. Reproduction occurs *locally*, within the same sequential process; however, migration among parallel subpopulations improves the population quality by injecting the best chromosome in the current sequential population. In what follows, we describe the local reproduction process while we describe in Sec. 4 the parallel population migration.

There are many approaches to selecting parent chromosomes for reproduction. We have attempted various common used techniques such as as *roulette wheel* and *tournament* selection. However, we have noticed that such reproduction techniques did not work very well with this type of layout problem. Hence, we have devised a special reproduction procedure that preserves a mix of "good" and "bad" where *bad* chromosomes are chromosomes that either represent partial solutions or chromosome with high fitness value. The following criteria was used in selecting chromosomes for reproduction:

(1) Choose 5% from the best chromosomes.
(2) Randomly choose 10% from "bad" chromosomes with density fitness less than 60% or more that 150% of the best chromosome. The density fitness is the ratio between fitness value and number of blocks available in the chromosome.[a] Note that a chromosome's fitness value can exceed 100% if the chromosome represents a partial solution.
(3) Choose 10% of chromosomes with density fitness values within [0.6, 1.5].
(4) Randomly choose 10% from all remaining bad chromosomes.
(5) Randomly choose 80% from those chromosomes with density fitness between 80% to 120% of the best chromosomes.
(6) Choose randomly the remaining chromosomes. Note that this step will only be used if the above categories fail to fill the new population with the given percentage.

3.6. *Coding of implementation*

To obtain a meaningful coding for the layout problem, one has to address the question on how to handle unfeasible placements that are suggested by the genetic algorithm. Obviously, if blocks positions are randomly chosen, a lot of them would be unfeasible. In general there are two different methods to handle these invalid implementations:

- Punishing unfeasible solutions with a prohibitive cost function.
- Repairing invalid solutions with a mechanism that incorporates domain knowledge.

[a]We use the density fitness value rather than the fitness value itself since some chromosomes may not have all the blocks in question.

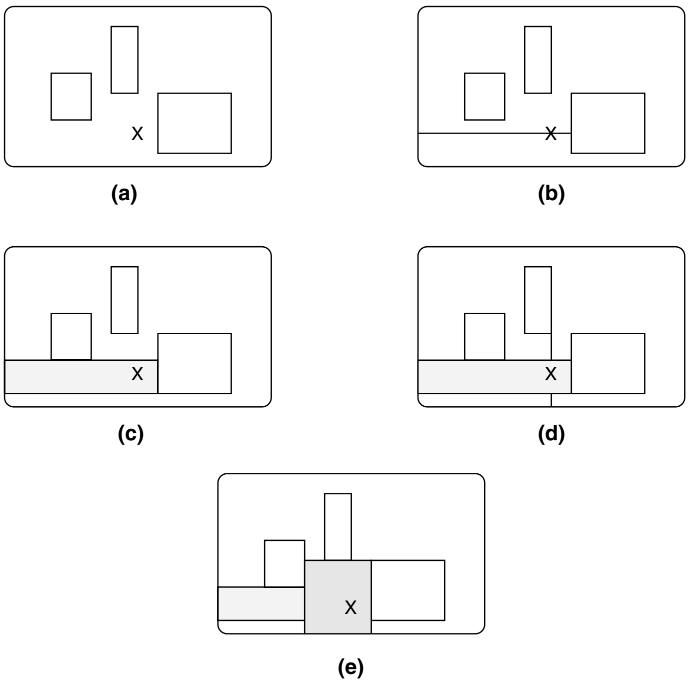

Fig. 2. *FindRegion* operator: (a) Initial insertion point, (b) find possible insertion range in the X-direction, (c) allowable region for insertion in the X direction, (d) find possible insertion range in the Y-direction, (e) final allowable insertion region.

In this work, we adopt a constructive approach. Thus, we limit the random points to feasible regions only by using a special operator *FindRegion*. The *FindRegion* operator determines if a given block can be placed at or in the neighborhood of a certain location $\mathrm{P}_{\langle x,y \rangle}$. In case where the algorithm results with unfeasible solutions, these solutions are repaired using special operators, *AddMissingBlocks* and *FixBlocks*.

3.6.1. *FindRegion*

FindRegion is a special operator that returns the range at P in both the X- and Y-direction where the block (B_i) with dimensions (L_i, W_i) can be positioned without overlapping with other blocks. If the *FindRegion* operator fails to return a range, then the operation is aborted. The *FindRegion* operator is very fast and efficient and has a run time complexity in the order of $O(w + h)$ where w represents the site width and h represents the site height.

FindRegion works as an *expanding balloon*. Thus, it blows in one direction and when it reaches a facility or the site's boundary, it switches to blow in the other direction. In order to track this *expanding balloon*, four arrays are needed; *left*, *right*, *up* and *down* that store the nearest distance of the neighbor block called *LArray*, *RArray*, *UArray* and *DArray*:

(1) Get the dimensions of the block to be inserted and the location where it is to be inserted. Check if there are any blocks at the insertion point, then the operation is aborted.
(2) Fill $LArray$ and $RArray$ with the distance to the nearest boundary, right and left of P. This results with a temporary block $B[1, XMax]$ at the insertion point.
(3) Fill the $UArray$ and $DArray$ with the distance to the nearest boundary, from above and below.
(4) Update B by shrinking to fit within the new neighbors on the upper bound.
(5) Update B by shrinking to fit within the new neighbors on the lower bound.
(6) The function returns $(x1, y1)$, $(x2, y2)$ that defines our range.

3.6.2. *AddMissingBlocks*

The *AddMissingBlocks* is a special operator that acts on chromosomes that have missing blocks. That is, blocks for which no feasible positions were found. This may happen as a result of an inefficient use of the site space. This operator attempts to find feasible positions for the missing blocks in a chromosome by applying a sequence of mutations.

3.6.3. *FixBlocks*

The FixBlocks is a special operator that repairs the positions of two blocks that have distance or orientation constraints between them.

3.7. *Genetic operators*

In order to explore the layout design space, we proposed various genetic operators that we describe next. The operators are applied iteratively if the maximum number of operators was not reached. After the operator is applied to a chromosome, the *chromosome age* is incremented if it is within 60% of the maximum allowed age.

Note that all operators use the special *FindRegion* operator in order to determine a feasible region where a block can be inserted without an overlap with other neighboring blocks. In the case of missing blocks, the chromosome is repaired using the *AddMissingBlocks* operator.

3.7.1. *Mutation*

The mutation operator selects a random location (X, Y) and a random block and attempts to insert the block at the location. If the block cannot be inserted after Op_n random attempts, i.e. no feasible insertion region is found then the operation is aborted.

3.7.2. *Inversion*

The inversion operator flips a block's position across a horizontal or vertical axis passing through the centroid of the site. The operator uses a first-to-fit strategy; thus, the block is inserted at the first feasible region found.

3.7.3. *Swap*

The swap operator exchanges the positions of a two randomly selected blocks in a chromosome. It is possible that either block may not fit in the place of the other block. If at least one block fits, then the other block is mutated or else, the operation is aborted. The operator has Op_n chances to select two different blocks before it is aborted.

3.7.4. *Move*

The movement operation has a dual optimisation role — *local* and *global*. However, the main objective of this operation is to apply a local optimisation around the already found solution rather than a global one.

The movement operator moves a randomly selected block in a chromosome d units in the x- or y-direction. The value of d is either a *unit* or a random value based on a *gaussian distribution*, depending on the age of the chromosome. In other words, the more the chromosome changes, the less the movement units are. A precondition for the selected chromosome is that it contains more than one block. The algorithm has Op_n chances to apply the move operator before aborting.

The movement operation has another important usage, *hill-climbing*. The hill-climbing is trigered if the algorithm is stuck at a local minima. Thus, if the ratio of the total area of the site to the sum of the area of the blocks is above a threshold value, $HillClimbing_{\text{Site}}$, then the algorithm accepts the first possible movement without any restrictions.

3.7.5. *Rotation*

The rotation operator extracts a block; rotates it and then inserts it around the previous point. The rotation operator rotates a block 90° anti-clockwise about its upper-left corner.

3.7.6. *Flip2Edge*

This operator is similar to inversion with one difference: the axis pass through the centroid of a randomly selected reference block and not the site. Note that if the algorithm is at local optimisation stage, i.e. the operation has changed to a best state for a maximum time, then the operation rotates the block around only one axis.

3.7.7. *Age*

The age operator destroys chromosomes whose fitness value did not change after x iterations where x is a user parameter. This operator is intended to destroy solutions that will not lead to the optimal one and allow young chromosomes to evolve into better solutions.

4. Parallel Genetic Placement Algorithm

We have implemented a parallel genetic algorithm for the facility layout problem as cooperating sequential genetic algorithms. The algorithm is based on a Single Program Multiple Data (SPMD) Model. The method uses message passing in order to allow various processors to operate independently on isolated subpopulations of the individuals, periodically sharing its "best" individuals. Processors are connected through an *Ethernet* network and use Parallel Virtual Machine (PVM).

4.1. *Parallel computational model*

In order to parallelise the placement algorithm, the population was partitioned into subpopulations that evolve independently using sequential GA. Interaction among subpopulations is allowed through *migration*. The parallel execution model is a more realistic simulation of natural evolution in which communities are isolated but occasionally interact through migration or cross-communities mating.[39]

Parallel migration implies that changes in a population come not only from inheriting portions of one's parents' genes, albeit with occasional random mutations, but also from the introduction of new species into the population. In nature, this movement between subpopulations is often a survival response that is responsible for several tasks including the selection and sending of emigrants in addition to the reception and integration of these immigrants. Note that population migration introduces communication overhead.

There are two common models for migration,[41,42] the *island model* where individuals are allowed to be sent to any other subpopulations and the *stepping stone model* that limits migration only to neighboring subpopulations. Both models have advantages and disadvantages. The obvious advantage is that, except for the communication of "best individuals" that occurs only once every k generations, both are embarrassingly parallel[b] in nature. Cohoon[6] observed that when a substantial number of individuals migrated between isolated subpopulations, new solutions were found shortly after the migration occurred.

In order to reduce the communication overhead while ensuring a global migration of individuals, we have adopted a hybrid approach that is based on the following. We allocate the sequential GA (slaves) to groups where the number of

[b] An embarrassingly parallel computation is an ideal parallel computation that can be immediately divided into completely independent tasks that can be executed simultaneously.

processes within a group are set by the user. Next, a group of processes share their best chromosomes every G_N generation. On the other hand, every L_N generations, the best chromosome is broadcast to all processes where $G_N \ll L_N$. This approach gives the genetic algorithm a chance to reconstruct different layouts starting from the best rather than from the initialised generations. Note that all processes are synchronised with the best solution through barriers.[c]

4.2. *Parallel algorithm*

The parallel model that we implemented is shown in Fig. 3, based on a SPMD master-slave model. The master algorithm, shown in Fig. 4, reads the site resources and constraints as well as the problem parameters. This includes the migration parameters as well as the number of slaves to be spawned. The master organises the slave processes into groups and distributes the subproblems among the slaves which do the actual computational work. The master is also responsible for migarting the best chromosome among various processes and performs some analysis before the final results are produced.

The slave algorithm, shown in Fig. 5, receives the data location as well as the constraints from the master. Every slave generates its own local initial population and executes its own sequential genetic algorithm. The GA operations are applied in the following sequence: *move*, *rotate*, *invert*, *mutate*, *swap*, and *Flip2Edge*. Every operator has a probability associated with it. The product of the operator's probability and the population size determines the maximum number of times this

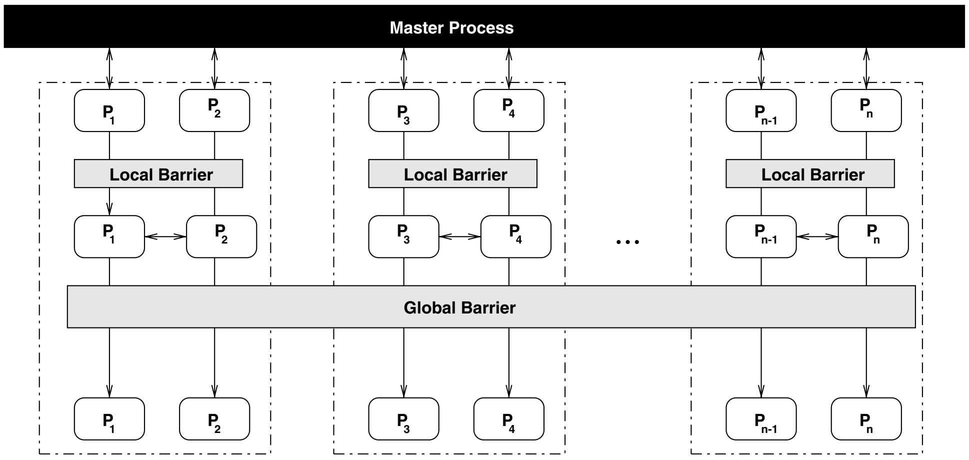

Fig. 3. Parallel genetic layout placement model.

[c]A barrier is a mechanism that prevents any process from continuing past a specified rendezvous point until all the processes are ready.

```
Master_Process(Site, Facilities, Constraints)
{
  ■ Get resources, constraints, population size (N) and nb. of generations (Ng).
  ■ Get the nb. of parallel slave processes to be spawned.
  ■ Spawn all slaves and send the necessary information.
  ■ forall (i = 0; i < number_slaves; i++)
      {
       if ($i == L_N$)
          local_migration();      /* Exchange the best chromosome among neighboring processes */
       else if ($i == G_N$)        /* Exchange the best chromosome among all processes */
          global_migration();
      }
}
```

Fig. 4. Master program.

operator can be fired in one generation. Operator's probabilities and values of other parameters were determined experimentally and are as shown in Table 1. At the end of each generation, the slave program computes the fitness value for each chromosome, finds the best chromosome and filters the population based on the age factor.

Every L_N iterations, local population stabilises. The algorithm introduces a new competition chromosome from neighboring processes (processes that are running on the same machine). Every G_N iterations, all slaves sends their best chromosome to the master process. The master then broadcasts this best chromosome to all slaves. In both cases, the best chromosome migration simulates a change in the environment and helps the subpopulation elements to rapidly evolve to adapt to this new change.

5. Experimental Results

A C++ program was developed on a dedicated Linux cluster of parallel machines at the Lebanese American University. We tested the parallel algorithm on three sets of layout problems that are designed to test the algorithm's performance as the total-object-to site area ratio is varied in the following cases: *equal-size with equal-weights objects, unequal-size with unequal-weights objects, unequal-size with unequal-weights objects and 2-dimensional constraints between objects.* In all these cases, the population size is set to 100 and the number of generations is limited to 400.

5.1. *PVM: Parallel Virtual Machine*

PVM[43] is a library of software routines used for cluster parallel programming. It provides a software environmnet for message passing between homogeneous and

```
Parallel_Genetic_Placement(Site, Facilities, Constraints)
{
  ■ Get resources, constraints, population size (N), nb. of generations (Ng),
  nb. of times an operator is applied and probabilities from the master.
  ■ Evaluate the fitness of chromosomes using equation 1. Keep the best
  for i = 0 to $N_g$ do
  {
    for (j = 0; $j <$ NumberOfOperators; j++)
    {
      Randomly select a chromosome from current_pop for mating.
      if (P < Pmove && MoveCount < MaxNbMove)
          Randomly select a block in the chromosome and get a random move unit
          based on a guassian probability and apply move
      if (P < Protation && RotationCount < MaxNbRotation)
          Randomly select a block in the chromosome and rotate
      if (P < Pinverse && InverseCount < MaxNblnverse)
          Randomly select a block in the chromosome and apply Operator Inverse
      if (P < P mutation && MutationCount < MaxNbMutation)
          Randomly select a block in the chromosome and apply mutation
      if (P < Pswap && SwapCount < MaxNbSwap)
          Randomly select two blocks in the chromosome and Apply Operator Swap
      if (P < PFlip2Edge && Flip2EdgeCount < MaxNbFlip2Edge)
          Randomly select a block in the chromosome and apply Operator mutation
    }
      ■ Repair the partial solution by adding all missing blocks;
      ■ Filter the population based on the age
      ■ Evaluate the cost and the fitness for each chromosome using equation 1.
      ■ Repeat the above for current_pop forming a new population, new_pop_.
      ■ Perform local reproduction
      ■ if ($i == L_N$)
     local_migration();      /* Exchange the best chromosome among neighboring processes
                                             through the master*/
      else if ($i == G_N$)
     global_migration();     /* Exchange the best chromosome among neighboring processes
                                             through the master*/
}
```

Fig. 5. Slave program.

Table 1. Parallel GA placement parameters.

Parameter	Value	Parameter	Value
Population size	200	Probability of Rotation	0.2
Probability of Mutation	0.1	Probability of Swap	0.1
Probability of Move	1	Probability of Flip2Edge	0.2
Probability of Inverse	0.1	Max Age	50
$HillClimbing_{\text{site}}$	0.05	Op_n	15
L_N	50	G_N	15

heterogenous computers and has a collection of library routines that the user can employ with C or Fortran. The problem is decomposed into several separate programs that run on different workstations. Programs are usually organised in a master-slave arrangement whereby the single master program is first executed and all others are spawned from this master process. Processes are automatically allocated to processors without any relationship to the number of processors.

We have built at the Lebanese American University a dedicated Linux cluster of parallel machines. The cluster is assembled in a closet and can only be remotely accessed. Currently, the cluster has eight Intel P-800 machines but we will be adding to this cluster those workstations that become surplus at LAU and thus the cluster is expected to grow. The cluster is organised into two different subnets for maximum performance. Every workstation on the cluster has two networks cards connected to two different switches (one for each LAN) thus every computer on the cluster has a simulated LAN segment to itself — the segment is busy only when a frame is being transferred to or from the computer. The cluster has been benchmarked using `netperf` and found to guarantee a 189 MBPS communication bandwidth between any two different machines. In other words, the cluster provides for a communication bandwidth in the order of a measured 1152 MBPS.

5.2. *Case 1: layout of equal-size with equal-weights objects*

In this section, the GA is used to solve bin-packing-like problems where objects have the same dimensions and have equal weights between them. No geometric constraints were considered between objects for problems in this case. The four problems selected and shown here illustrate the GA's performance in solving problems with different number of objects to be laid out (10, 25, 45, and 60 objects) and thus different total-objects-to-site-area ratios (OSAR). Table 2 shows the input used with the four problems of Case 1.

Table 3 shows a comparison between the fitness value of the layouts generated by the GA along with the run-time of each problem and the optimal or best layouts found. The GA solutions found are on average within 2% from the optimal solution. This is considered to be a very good result given the size of the problems solved.

Table 2. Input for case 1 problems.

Problem	Number of Objects	Dimensions of Objects	W_{ij}, $\forall_i$, and $\forall_j$	Total Object to Site Area Ratio (OSAR) (%)
1	10	2×1	1	10
2	25	2×1	1	23
3	45	2×1	1	40
4	60	2×1	1	54

Table 3. Comparison between GA sequential, parallel GA and best solution for case 1 problems.

Problem	Sequential GA Fitness	Parallel GA Fitness	Optimal or Best Fitness	OSAR (%)	Error (%)
1	135	131	131	10	0
2	1,438	1,438	1,407	23	2.20
3	6,226	5,852	5,852	40	0
4	12,389	12,389	12,145	54	2.00

5.3. *Case 2: layout of unequal-size with unequal-weights objects*

In this section, the GA is used to solve layout problems with layout objects of different dimensions and having different weights between them (i.e. with different levels of interactions between them). No geometric constraints were considered for problems in this case as well. The four problems selected and shown here have the same total-object-to-site-area ratios as the four problems of case 1. This choice is intentional to test how the GAs performance varies when layout objects are not all equal. The inputs associated with the four problems are as shown in Table 4 below.

Table 4. Input for case 2 problems.

Problem	Objects	$L \times W$	Number of Objects	W_{ij}	OSAR (%)
1	1	4×2	1		10
	2,3,4,5	2×2	4	5; $\forall i = 2, \ldots, 5$ $\forall j = i + 1 = 2, \ldots, 5$	
				20; $i = 1$ and $j = 2$	
2	1,2,3	4×2	3	10; $\forall i = 1, \ldots, 3$ $\forall j = i + 1 = 1, \ldots, 3$	23
	4,5,6,7,8,9,10	2×2	7	5; $\forall i = 4, \ldots, 10$ $\forall j = i + 1, \ldots, 10$	
				20; $i = 1$ and $j = 4$ $i = 2$ and $j = 5$	
3	1,2,3,4,5,6,7	4×2	7	10; $\forall i = 1, \ldots, 7$ $\forall j = i + 1, \ldots, 7$	40
	8,9,10,11,12	2×2	8	5; $\forall i = 8, \ldots, 15$ $\forall j = i + 1, \ldots, 15$	
				20; $i = 1$ and $j = 8$ $i = 2$ and $j = 9$ $j = 3$ and $i = 10$	
4	1,2,3,4,5,6,7 8,9,10	4×2	10	10; $\forall i = 1, \ldots, 10$ $\forall j = i + 1, \ldots, 10$	54
	11,12,13,14,15, 16,17,18,19,20	2×2		5; $\forall i = 11, \ldots, 20$ $\forall j = i + 1, \ldots, 20$	
				10; $i = 1$ and $j = 11$ $i = 2$ and $j = 12$ $i = 3$ and $j = 13$	

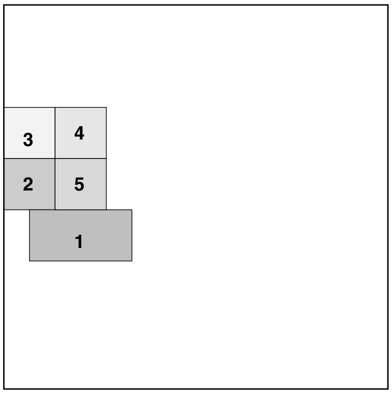

Fig. 6. The parallel GA solution of case 2, problem 1 (Table 4).

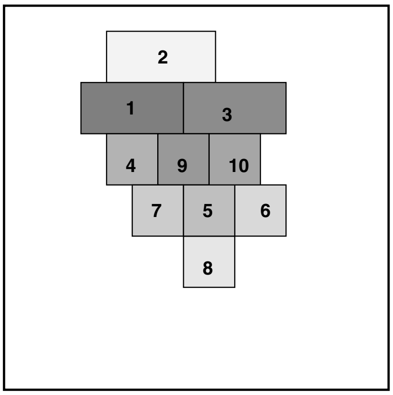

Fig. 7. The parallel GA solution of case 2, problem 2 (Table 4).

Figures 6–8 show the layout solutions returned by the GA while Fig. 10 illustrate the convergence speed for problem 2 where P1, P2, P3, and P4 denote four parallel GA solutions while P0 corresponds to the sequential solution. Note the variation in convergence speed among parallel processes due to chromosomes migration.

Table 5 shows a comparison between the fitness value of the layouts generated by the GA along with the run-time of each problem and the optimal or "best" layouts found. The GA solutions for problems with OSAR less than 50% are on average within 6% from the optimal solution. The "goodness" of the solution drops suddenly for problem 4 with OSAR 50% as it is illustrated by the solution of problem 4.

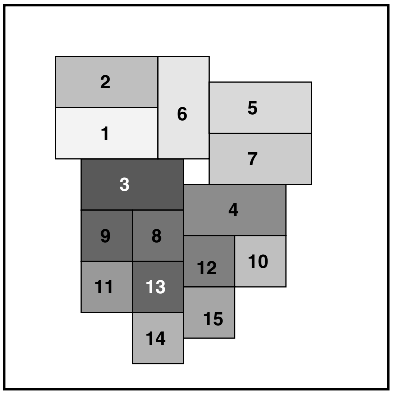

Fig. 8. The parallel GA solution of case 2, problem 3 (Table 4).

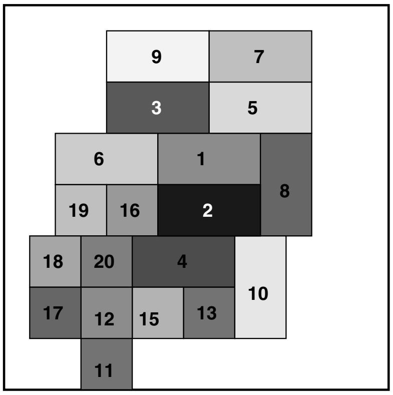

Fig. 9. The parallel GA solution of case 2, problem 4 (Table 4).

Table 5. Comparison between GA sequential, parallel GA and best solution for case 2 problems.

Problem	Sequential GA Fitness	Parallel GA Fitness	Optimal or Best Fitness	OSAR (%)	Error (%)
1	130	120	120	10	0
2	600	600	580	23	3.44
3	1,880	1,880	1,760	40	6.88
4	4,840	4,620	3,950	54	16.96

5.4. *Case 3: layout of unequal-size with unequal-weights and 2-dimensional constraints between objects*

Finally, this section treats the class of layout problems where objects are of different size and have variable weights between them and have geometric constraints on their relative positions in addition to the default non-overlap constraints. Only two problems are presented in this section. These problems are intended to test the

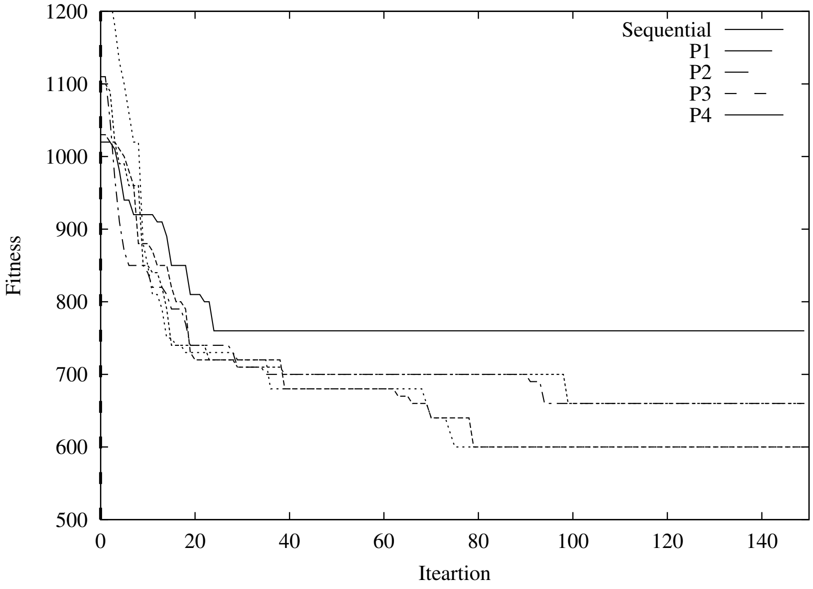

Fig. 10. Convergence speed in case 2, problem 2. Note that P1, P2, P3, and P4 correspond to four parallel GA solutions while P0 corresponds to the sequential process. Note the variation in convergence speed among parallel processes duo to chromosomes migration.

Table 6. Input data for the two problems of case 3.

Object	(L, W)	Area	Object	(L, W)	Area
Cons. Site	(30, 20)	600	Object 6	(9, 4)	36
Object 1	(7, 5)	35	Object 7	(9, 5)	45
Object 2	(4, 4)	16	Object 8	(10, 4)	40
Object 3	(8, 5)	40	Object 9	(2, 2)	4
Object 4	(2, 1)	2	Object 10	(4, 4)	16
Object 5	(7, 7)	49			

Table 7. Proximity weights for the two problems of case 3.

Object$_i$	Object$_j$	W_{ij}	Object$_i$	Object$_j$	W_{ij}
Object 1	Object 2	25	Object 6	Object 7	100
Object 2	Object 3	50	Object 8	Object 9	25
Object 4	Object 5	50	Object 9	Object 10	25

GAs performance in solving problems with a relatively smaller number of layout objects but with a larger variety of geometric constraints between them. The layout objects dimensions and proximity weights between them for both problems are given in Tables 6 and 7 respectively. Both problems have an OSAR value of 50% approximately.

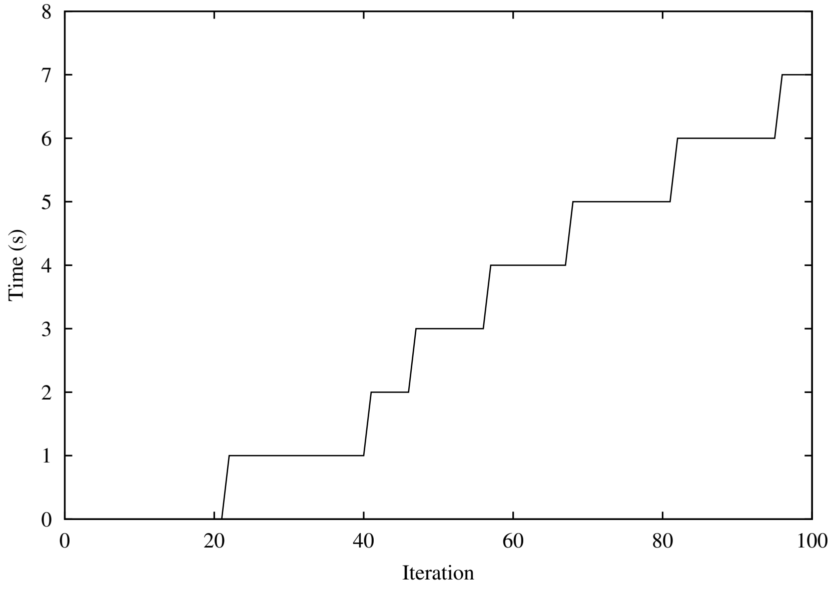

Fig. 11. Communication time for case 2, problem 2.

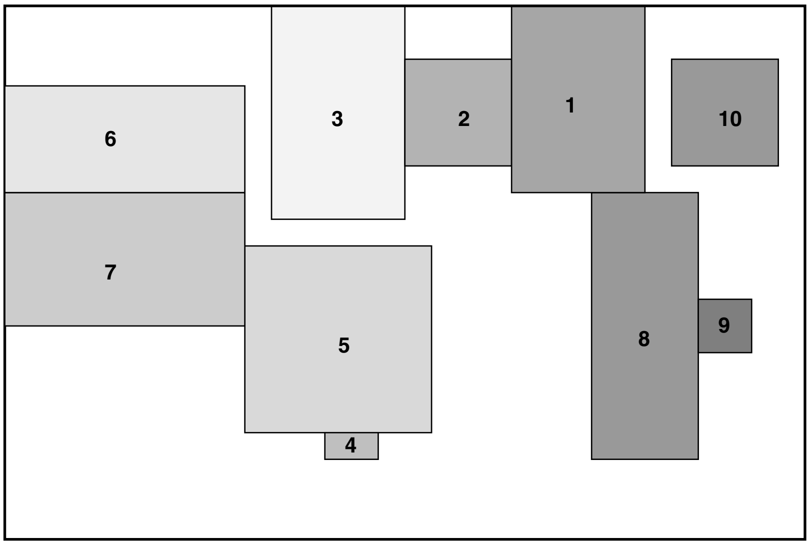

Fig. 12. The parallel GA solution of case 3, problem 1 (Table 6).

The first problem has in addition to the weights defined in Table 7, the following geometric constraints between objects: *object 6* is constrained to be to the North of *object 7*, and *object 9* and *object 10* should have a minimum distance of 5 units in the Y-direction between them. The parallel GA solution to the above problem is shown in Fig. 12 and has a fitness value of 1,300. Note that the gray-shaded

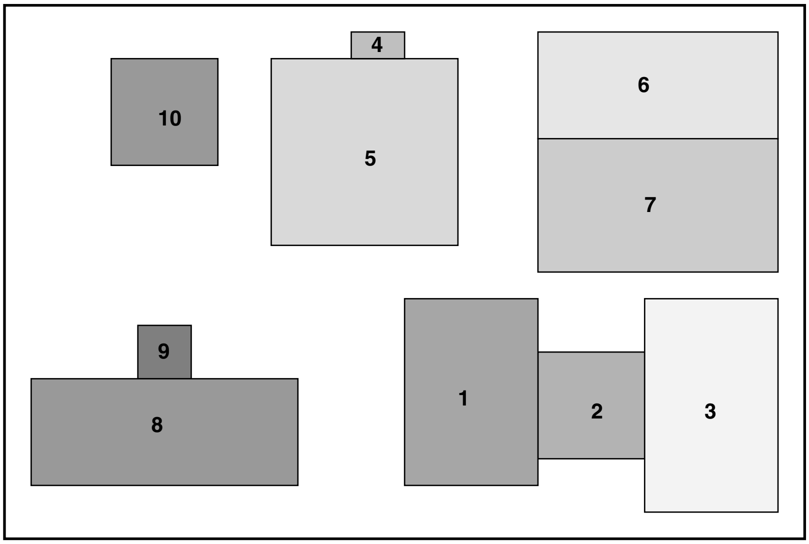

Fig. 13. Optimal solution of case 3, problem 1 (Table 6).

Table 8. Comparison between GA sequential, parallel GA and best solution for case 3 problems.

Problem	Sequential GA Fitness	Parallel GA Fitness	Optimal or Best Fitness	OSAR (%)	Error (%)
1	1,325	1,300	1,262.5	47	2.97
2	1,350	1,300	1,262.5	47	2.97

rectangle represents *object 4*. The optimal solution for this problem has a fitness value of 1,262.5. There are alternative optimal layouts for this problem. Figure 13 shows only one such optimal layout.

The second problem considered has the same objects and relationships among them as the previous one. It has however two additional geometric constraints in addition to the ones defined in problem 1: a maximum distance constraint of 8 units in the Y-direction between *object 5* and *object 7* and a minimum distance constraint of 5 units in the Y-direction between *object 1* and *object 10*. Table 8 summarises the results of this case. As it can be seen, the GA found solutions for problems with OSAR of 50% that are within 3% of the optimal solution. The algorithm converged to the best solution in this case in less than 75 iterations as shown in Figs. 14 and 16 while the communication time was kept to minimal (Figs. 15 and 17).

6. Conclusion

This paper presented a parallel genetic algorithms for solving the geometrically constrained site layout problem. Key features of this algorithm are that it uses a

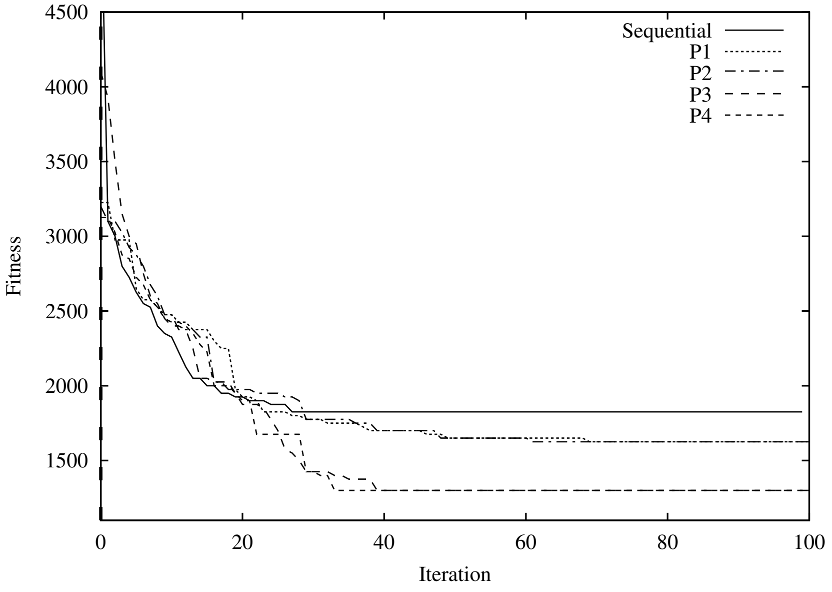

Fig. 14. Convergence speed in case 3, problem 1.

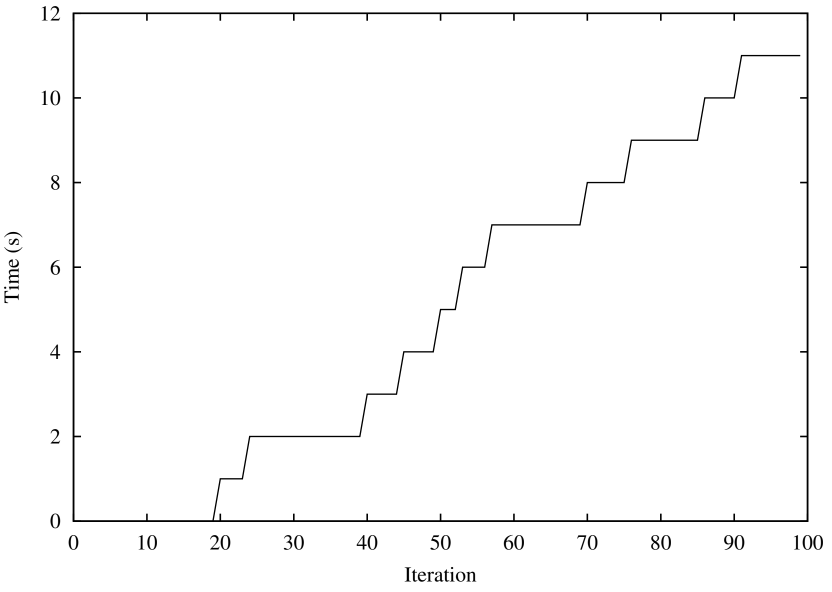

Fig. 15. Communication time for case 3, problem 1.

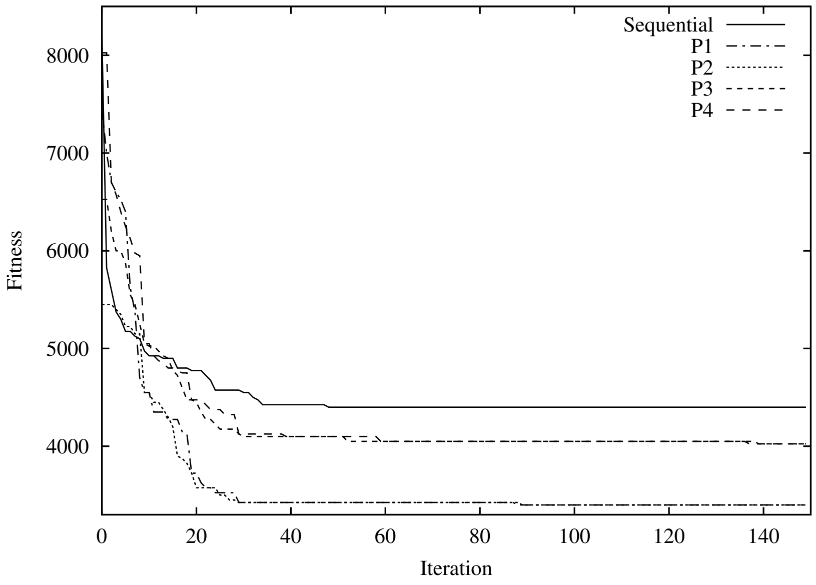

Fig. 16. Convergence speed in case 3, problem 2.

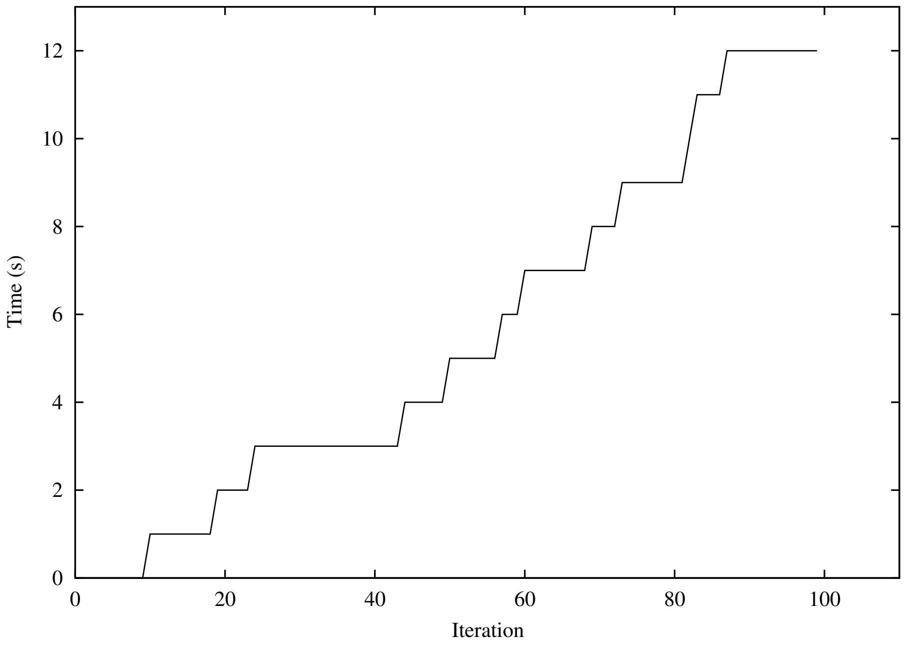

Fig. 17. Parallel communication time for case 3, problem 2.

large number of different GA operators to vary positions of objects around the site and that it maintains in each generation chromosomes representing partial layout solutions.

The algorithm was tested on different problems with a different number of blocks and constraints. In most attempted cases the algorithm returned close to optimal solutions in a reasonable time (less than 2 minutes) after 200 generations. Finally, the relationship between computational time and the number of layout objects was found to be approximately linear.

References

1. M. Garey and D. Johnson, *Computers and Intractability — A Guide to the Theory of NP-Completeness* (W. H. Freeman and Company, 1979).
2. S. Sahni and T. Gonzalez, P-complete approximation problem, *J. Assoc. Comput. Mach.* **23**(3) (1976) 555–565.
3. G. C. Armour and E. S. Buffa, A heuristic algorithm and simulation approach to relative allocation of facilities, *Management Sci.* **9** (1963) 294–309.
4. J. Cohoon and W. Paris, Genetic placement, in *Proc. IEEE Int. Conf. CAD* (1986), pp. 422–425.
5. J. Cohoon, S. Hedge, W. Martin and D. Richrads, Distributed genetic algorithms for the floorplan design problem, *IEEE Trans. CAD* **10**(4) (1991) 483–492.
6. J. Cohoon, S. Hedge, M. Martin and D. Richards, Punctuated equilibria: A parallel genetic algorithm, in *Proc. Second Int. Conf. Genetic Algorithm* (New Jersey, New York, 1987).
7. A. Kusiak and S. S. Heragu, The facility layout problem, *Eur. J. Operat. Res.* **29** (1987) 229–251.
8. R. D. Meller and K.-Y. Gau, The facility layout problem: Recent and emerging trends and perspectives, *J. Manufacturing Syst.* **15**(5) (1996) 351–366.
9. P. Kouvelis and W. Chiang, Optimal and heuristic procedures for row layout problems in automated manufacturing systems, *J. Operat. Res. Soc.* **47**(6) (1996) 803–816.
10. Y. C. Ho and C. L. Moodie, Machine layout with a linear single-row path in an automated maufacturing system, *J. Manufacturing Syst.* **17**(1) (1998) 1–22.
11. Y. A. Bozer and S. C. Rim, Branch and bound method for solving the bi-directional circular layout problem, *Appl. Math. Modeling* **20**(5) (1996) 342–351.
12. D. J. van Camp, M. W. Carter and A. Vannelli, A nonlinear optimization approach for solving facility layout problems, *Eur. J. Operat. Res.* **57** (1991) 174–189.
13. P. C. Gilmore, Optimal and suboptimal algorithms for the quadratic assignment problem, *J. Soc. Ind. Appl. Math.* **10** (1962) 305–313.
14. E. L. Lawler, The quadratic assignment problem, *Management Sci.* **13** (1963) 42–57.
15. E. Feng, X. Wang, X. Wang and T. Honfgei, An algorithm of global optimization for solving layout problems, *Eur. J. Operat. Res.* **114** (1999) 430–436.
16. K. Y. Tam and S. G. Li, A hierarchical approach to the facility layout problem, *Int. J. Prod. Res.* **29**(l) (1991) 165–l84.
17. B. N. K. Ann and C. K. Chua, Knowledge-based system for strip layout problem, *Comput. Ind.* **25**(1) (1994) 31–44.
18. S. S. Heragu and A. Kusiak, Machine layout: an optimization and knowledge-based approach, *Int. J. Prod. Res.* **28**(4) (1990) 615–635.
19. Y. K. Chung, Neuro-based expert system for facility layout construction, *J. Intel. Manufacturing* **10**(5) (1999) 359–385.

20. P. Kouvelisy and W. Chiang, Simulated annealing approach for single row layout problems in flexible manufacturing systems, *Int. J. Prod. Res.* **30**(4) (1992) 717–732.
21. K. Y. Tam, Genetic algorithms, function optimization, and facility layout design, *Eur. J. Operat. Res.* **63** (1992) 322–346.
22. K. Tsuchiya, S. Bharitkar and Y. Takefuji, A neural network approach to facility layout problems, *Eur. J. Operat. Res.* **89** (1996) 556–563.
23. J. S. Kochhar and S. S. Heragu, Facility layout design in a changing environment, *Int. J. Prod. Res.* **37** (1999) 2429–2446.
24. M. Rajasekharan, B. A. Peters and T. Yang, Genetic algorithms for facility layout design in flexible manufacturing systems, *Int. J. Prod. Res.* **36**(1) (1998) 95–110.
25. J. Garces-Perez, D. A. Schenefeld and R. L. Wainwright, Solving facility layout problems using genetic programming, *Proc. First Ann. Conf. Genetic Program.* (MIT Press, 1996), pp. 182–190.
26. K. Y. Tam and S. G. Li, A simulated annealing algorithm for allocating space to manufacturing cells, *Int. J. Prod. Res.* **30**(1) (1992) 63–87.
27. D. M. Tate and A. E. Smith, Genetic algorithm optimization applied to variations of the unequal area facilities layout problem, in *Proc. 2nd Industrial Eng. Res. Conf.*, 1993, Los-Angeles, CA, pp. 335–339.
28. D. M. Tate and A. E. Smith, A genetic approach to the quadratic assignment problem, *Comput. Operat. Res.* **22**(1) (1995) 73–83.
29. K. Kado, P. Ross and D. Corne, A study of genetic algorithm hybrids for facility layout problems, *Proc. First Annl. Conf. Genetic Program.* (1996), pp. 182–190.
30. I.-C. Yeh, Construction-site layout using annealed neural network, *J. Comput. Civil Eng.* ASCE **9**(3) (1995) 201–208.
31. H. Li and P. E. Love, Site-level facilities layout using genetic algorithms, *J. Comput. Civil Eng.* ASCE **12**(4) (l989) 227–231.
32. I. Tommelein, R. Levitt and T. Confrey, Sight plan experiments: Alternate strategies for site layout design, *J. Comput. Civil Eng.* ASCE **5**(1) (1991) 42–63.
33. M.-Y. Cheng, *Automated Site Layout of Temporary Facilities Using Geographic Information Systems (GIS)*, PhD. Dissertation, Civil Engineering Department, University of Texas, Austin (1992).
34. P. Zouein and I. Tommelein, Dynamic layout planning using a hybrid incremental solution method, *J. Const. Eng. Management*, ASCE **125**(6) (1999) 400–408.
35. S. Cheung, K. T. Thomas and C. Tam, Site pre-cast yard layout arrangement through genetic algorithms, *Automation Constr. J.* **11** (2002) 35–46.
36. H. Harmanani, P. Zouein and A. Hajar, An evolutionary algorithm for solving the geometrically constrained site layout problem, in *Proc. ASCE Int. Conf. Comput. Civil Building Eng. (ICCCBE VIII)*, Stanford University, Stanford, CA, Vol. 2, pp. 1442–1449.
37. P. Zouein, H. Harmanani and A. Hajar, Genetic algorithm for solving site layout problem with unequal-size and constrained facilities, *J. Comput. Civil Eng.* ASCE **16**(2), April 2002.
38. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning* (Addison-Wesley, Boston, 1989).
39. S. M. Sait and H. Youssef, *Iterative Computer Algorithms with Applications in Engineering* (IEEE Computer Society Press, 1999).
40. D. E. Goldberg, Sizing populations for serial and parallel genetic algorithms, in *Proc. Third Int. Conf. Genetic Algorithms*, ed. J. D. Schaffer (Morgan Kaufman, San Mateo, CA, 1989).

41. A. Chipperfield and P. Fleming, Parallel genetic algorithms, in *Parallel and Distributed Handbook* (McGraw Hill, NY, 1996).
42. B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers* (Prentice Hall, NJ, 1999).
43. A. Geist, J. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine* (MIT Press, Cambridge, MA, 1994).