ELSEVIER

Advances in Engineering Software Vol. 29, No. 10, pp. 763–786, 1998
© 1998 Published by Elsevier Science Ltd
Printed in Great Britain. All rights reserved
PII: S0965-9978(97)00062-8
0965-9978/98/$19.00+0.00

Parallel processing, neural networks and genetic algorithms

B.H.V. Topping*, J. Sziveri, A. Bahreinejad, J.P.B. Leite & B. Cheng

Department of Mechanical and Chemical Engineering, Heriot-Watt University, Edinburgh, U.K.

In an earlier paper[1] some recent developments in computational technology to structural engineering were described. The developments included: parallel and distributed computing; neural networks; and genetic algorithms. In this paper, the authors concentrate on parallel implementations of neural networks and genetic algorithms. In the final section of the paper the authors show how a parallel finite element analysis may be undertaken in an efficient manner by preprocessing of the finite element model using a genetic algorithm utilizing a neural network predictor. This preprocessing is the partitioning of the finite element mesh into sub-domains to ensure load balancing and minimum interprocessor communication during the parallel finite element analysis on a MIMD distributed memory computer. © 1998 Published by Elsevier Science Limited. All rights reserved.

1 INTRODUCTION

Design in many branches of engineering requires linear and non-linear finite element analysis. An ever increasing requirement to explore more design possibilities results in large-scale finite element computations which may be part of an optimization process or a simulation of an artefacts behaviour. Parallel and distributed computing permits the engineer to undertake the finite element analysis in a considerably shorter time, reducing the time between development of design concept and production of the final artefact.

Application of parallel and distributed computing to finite element analysis requires the development of parallel algorithms. For these algorithms to be applied effectively then considerable preprocessing of the finite element model is required to ensure that the analysis is efficiently undertaken in parallel. New and emerging technology in this area includes neural networks and genetic algorithms. This paper briefly discusses parallel and distributed computing and then parallel neural networks and parallel genetic algorithms. Finally, it is demonstrated how these techniques may be utilized in the preprocessing for a parallel non-linear transient finite element analysis.

The concepts discussed here result in the generation of finite element meshes in parallel permitting the finite element discretization to be generated in parallel without ever having to be assembled on a master or root processor.

Structural analysis and design for large-scale problems requires considerable computational effort. The implementation of parallel algorithms for structural engineering problems has been the subject of much research recently.[2] Parallel algorithms may be generally classified as being of one of three types:

- Processor farming: in this type of algorithm each processor in the network executes code in isolation from all other processors except for a master or root processor. This type of parallelization is also called independent task, task farming or event parallelism. With processor farming interprocessor communication is not generally permitted.
- Geometric parallelism: in which each processor executes the same codes on data corresponding to a sub-region or sub-domain being simulated or processed. Interprocessor communication is possible to permit the exchange of boundary data between neighbouring processors representing connections between the sub-domains.
- Algorithmic parallelism: in which each processor is responsible for part of the algorithm and the data passes through each processor in turn. Different parts of the algorithm are executed on each processor. The processors represent a conveyor belt on which computation is performed. The computational load on each processor must be the same or balanced if a bottleneck is not to develop.

With processor farming and geometric parallelization the processors must all complete their tasks at the same time if

*Author to whom all correspondence should be addressed.

processors are not to be left idle waiting for others to complete their tasks. This requires load balancing of the computations which is an important aspect of parallel computations which will be discussed later. Parallel processing provides increased computational power for structural analysis by using a number of different strategies. Generally, these strategies may be classified as follows:

- The analysis problem may be sub-divided by geometrically dividing the idealization into a number of sub-domains. Since each of these domains will be subject to the same instructions, processor farming or geometric parallelization techniques may be used. This is described as an explicit domain decomposition approach.
- Alternatively, the system of equations for the whole structure may be assembled and solved in parallel without recourse to a physical partitioning of the problem. This is described as an implicit domain decomposition approach.

The explicit domain decomposition approach requires the finite element mesh to be split into a number of sub-domains using a domain decomposition algorithm. Domain decomposition approaches may be described as divide and conquer algorithms, since their object is to divide the larger problem into a series of smaller sub-problems. There are numerous techniques for domain decomposition; however, two basic approaches are common:

1. Iterative techniques: are used to solve the whole problem iteratively and information concerning nodes common to two or more domains must be communicated between processors after each iteration.
2. Sub-structuring techniques have been established for over 30 years. In this approach the sub-domains are treated as complex structural elements and the formulation for each sub-domain is carried out by only taking into account the degrees of freedom of the boundary nodes. Once the displacements of the boundary nodes have been determined the displacements within each substructure may be determined independently.

This paper concentrates on the iterative approach which is appropriate for non-linear transient finite element analysis. For the implementation of a parallel algorithm it is necessary that the given domain of the problem is discretized into a finite number of subregions or sub-domains. In this regard, it is important that the domain decomposition algorithm should be able to: handle irregular mesh geometries of arbitrarily shaped domains to make the method completely general; and to minimize the interface problem by providing partitioning interfaces which deliver minimum boundary node connectivities. This aids the reduction of the magnitude of the boundary problem and in physical terms reduces the communication overheads in the parallel computing system; while the sub-domains should carry approximately equal amounts of computational load. This is again to satisfy the physical requirement of the system ensuring that all the processors finish their work at about the same time and that the controlling processor is not forced to wait on account of a lagging processor. Mesh decomposition algorithms are discussed in detail elsewhere.[2]

The range of parallel computer architectures is discussed with respect to structural design in Refs[1,2], but the dominant computer architecture is currently MIMD (multiple instruction multiple data) with distributed memory. This classification corresponds to a wide range of supercomputers, high-performance multiple processor computers and loosely coupled networks of workstations.

1.1 Distributed and heterogeneous computing

The role of distributed computing is increasing because personal computers and workstations may now be networked providing engineers with more powerful integrated systems, without the additional expense of high-performance computers. These networks may be local area networks (LANS) or wide based area networks (WANs). They usually utilize the TCP/IP protocol with various types of network carrier including: thin ethernet coaxial cable; fibre optics; twisted pair cable; and ISDN links. These networks are usually heterogeneous in the sense that all the connected processors will not usually be of the same type or power. Some nodes of the network may themselves be MIMD or SIMD (single instruction multiple data) computers which represent high-performance compute nodes within the network. It is important to ensure that any computational task is allocated to the appropriate processor within the heterogeneous network.

1.2 PVM

Parallel virtual machine (PVM)[3] is a portable message passing programming system designed to link separate Unix host machines to create a virtual machine which is a single manageable computing resource. The PVM system is composed of two parts. The first is a daemon program which runs on all machines that are part of the virtual machine. This permits the user to run a PVM application at a Unix prompt from any of the machines. The second part of the system is a library of PVM interface routines. This library contains user callable routines for passing messages, spawning processes, coordinating tasks and modifying the virtual machine. Application programmes can be written in a mixture of C, C + + and FORTRAN, but must be linked with the PVM library.

1.3 Message passing interface

Message passing interface (MPI)[4] is a specification for a library of routines to be called from C and FORTRAN programs. This system provides an API (application programming interface) which permits a standard to develop parallel programmes with standard message passing. This interface

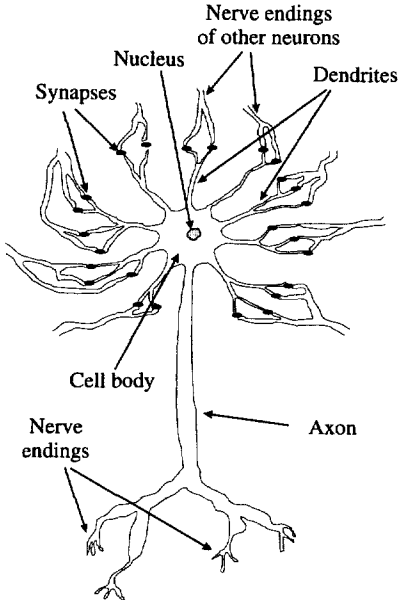

Fig. 1. Idealized representation of a biological neuron.

is a standard and a carrier such as PVM or P4 is required for distinct implementations if the operating system does not support the message passing. Both native and layered implementations are known for a variety of systems.

2 PARALLEL NEURAL NETWORKS

Artificial neural networks (ANNS) are computational models which attempt to mimic the learning function of the brain. The brain is the most complex system comprising billions of neurons. Each neuron is a processing unit which receives, processes and sends information. A biological neuron is made up of three parts: the cell body; the axon; and the dendrites. A typical biological neuron is shown in Fig. 1. Signals travel through the axon of other neurons, thus reaching the dendrites of another neuron. The signals then travel through a junction between the axon and the dendrites called the synapse. The response of a neuron depends on several biological and chemical factors corresponding to the synapses and the receiving neuron. A neuron may fire a signal if the magnitude of the signal is strong enough to activate it. The synaptic efficiency or strength is modified to adjust to the received signal. The brain is said[5] to learn when the synapses adjust themselves to accommodate to receive new signals.

Artificial neural networks are composed of a set of neurons or processing units which are connected together by means of connecting weights. ANNs which are structured to learn and generalize so that the network may learn by continuous adjustment of the weights of the connections.

Application of ANNs have received particular attention in the field of robotics, vision recognition, military and space exploration.[6–11] One of the drawbacks of ANNs is

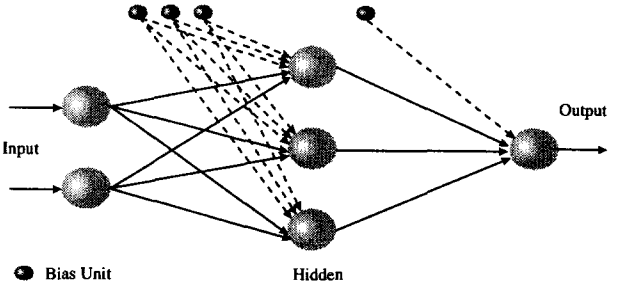

Fig. 2. A simple three-layered fully connected network.

that they are computationally expensive. This drawback may largely depend on the size of a network or the quantities of data involved. Despite the efforts that have been taken to improve the training algorithms to make them more efficient particularly in terms of computation time,[12–14] the greatest scope for improving the computational efficiency may lie with the parallel nature of ANNs algorithms. This nature enables the algorithms to be implemented on parallel architectures.

2.1 Backpropagation neural networks

A backpropagation (BP) network has a multilayered topology which is shown in Fig. 2. TH BP network consists of a set of neurons or units organized into a sequence of layers with full or patterned connecting weights between the layers. Biases may also be employed which act as weights from imaginary neurons that have an output of one at all times. Typically, only two layers terminate a network: the input layer to which data is presented for the network, and the output layer where the response of the network to a given input is received. Layers other than these two are called hidden layers.

Neural networks recognize patterns after training by the presentation of a number of input–output (i–o) patterns. The purpose of training a backpropagation[15] network is to adjust a set of initially randomized weights until a non-linear and continuous function representing a mapping space which includes the training patterns may be formed. Such networks learn the mapping task by using examples and hence no predefined knowledge of the problem is required. This is done when a set of inputs is presented to the input layer units. The inputs are propagated forward through to the other layers where they are multiplied by the associated weights. The sum of these products within each receiving unit is subjected to a transfer function which is non-linear, non-decreasing and differentiable. Commonly a sigmoid function is adopted as the transfer function. The result is an output which becomes the input to the next layer. Fig. 3 shows the input and output from a unit.

At the output layer, the computed output is compared with the desired output and an error value is computed. This error value together with the derivative of the transfer function are used to compute the error signal(s) for the output layer unit(s). The error signal(s) from the output layer

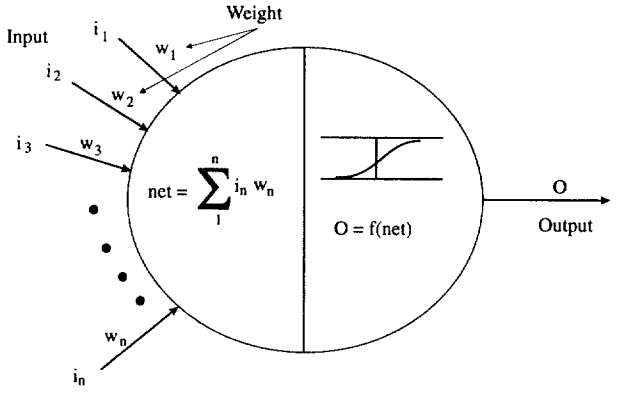

Fig. 3. The operation of a single artificial neuron.

unit(s) propagate back in order to calculate the error signal(s) for the previous hidden layer(s).[15] Once all the appropriate error signals have been computed they are used to calculate the delta-weights representing the quantity for which the corresponding weights should change. The learning process corresponds to performing gradient descent on a surface in weight space whose height at any point in weight space is equal to the measured error. The training process is repeated for all the training patterns until a desired root-mean-square (RMS) error level over all training patterns is achieved.[16] The RMS value of the training is given by:

$$\mathrm{RMS}_{\mathrm{err}} = \sqrt{\frac{1}{PN_{\mathrm{output}}} \sum_{p}^{P} \sum_{output}^{N_{\mathrm{output}}} (t_p - o_p)^2} \qquad (1)$$

where P is the total number of training patterns, p is a single training pattern, N_{output} is the number of units in the output layer, t_p is the desired output in the training data and o_p is the output of the network.

Training may be undertaken using either single or batch pattern training where:

- Single pattern training: the weights are modified after each i–o pattern has been presented to the net.
- Batch pattern training: the weights are changed after the presentation of a batch or all i–o patterns.[16,17]

2.2 Parallel BP neural nets

Parallel implementation of multi-layered neural networks may be carried out in a number of forms including:

- distribution of the network by dividing the units and or layers amongst the processors;
- distribution of units by representing each unit with a single processor; and
- distribution of i–o data patterns among the processors.[16,17]

The first method suggests that layers and/or units are distributed amongst the processors. This methods is particularly efficient if the constructed network consists of many units and layers, but if during the learning a large quantity of data has to travel through the pipeline then the method will be inefficient.

The second paralellization method is to represent each unit with an individual processor. Except for small network structures, an efficient implementation of this method may not be possible. This is due to the limited number of hardware links available in some parallel computers and the difficulty of undertaking point-to-point routing. Any point-to-point routing implementation will not contribute to the efficiency because the communication overhead would significantly increase.

If the network structure is not large, but a large number of training patterns is to be used, then the third method is likely to be more efficient. An example of parallel training of BP nets using the third approach may be found in Refs[16,17]. These references examine the parallel training of BP nets using single and batch training approaches. It was concluded[16] that the single pattern implementation requires a much higher degree of inter-processor communication. Batch pattern training may appear more appropriate for parallel training; however, a much slower rate of convergence generally results with this form of training. The overall efficiency may be impaired, although the training patterns may be presented to the network a far greater number of times than would required with single pattern training.

2.3 Parallel mean field annealing

Combinatorial optimization problems arise in many areas of science and engineering. Most computational solution methods that have been developed which generally yield good solutions to these problems rely on heuristics of some form. ANNs make use of highly interconnected networks of simple neurons or units which may be programmed to find approximate solutions to these problems.[18–20] They are also highly parallel systems and have significant potential for parallel hardware implementations.

The origin of the optimization neural network goes back to the work by Hopfield and Tank[21] which was a formulation to solve the travelling salesman problem (TSP). The Hopfield network is a feedback type of neural network where the output(s) from a processing unit is (are) fed back as the input(s) of other units through their interconnections. Fig. 4 illustrates a typical feedback architecture and Fig. 5 shows a simple representation of a Hopfield network.

Following the poor performance of Hopfield networks in determining valid solutions for the TSP problem, there followed considerable research effort to improve the performance of this type of network and to find ways of applying it to other optimization problems. At about the same time of the emergence of Hopfield networks, a new optimization method called simulated annealing[22] was developed. This technique provides a method for finding good solutions to most combinatorial optimization problems; however, the algorithm takes a long time to converge. To overcome this

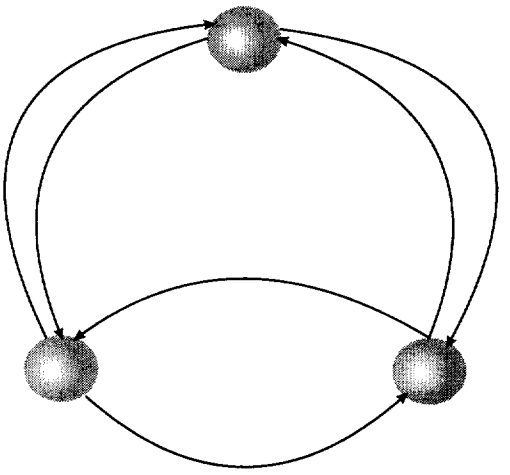

Fig. 4. A typical feedback/recurrent architecture.

problem, mean field annealing (MFA) networks were proposed as an approximation to simulated annealing. MFA is a deterministic approach which essentially replaces the discrete degrees of freedom in a simulated annealing problem with their average values as computed by the mean field approximation. The components of MFA networks include the following.

2.3.1 Hopfield network

NP problems may be mapped onto the Hopfield discrete state neural network using the Liapunov function which is defined as:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} t_{ij}s_i s_j + \sum_{i=1}^{N} I_i s_i \tag{2}$$

where: I_i is the bias of the ith unit; N is the number of processing units; s_i and s_j represent the state of the units i and j in the network connected through the weight t_{ij}. It is assumed that the t_{ij} matrix is symmetric and has no self-interaction (i.e. $t_{ii} = 0$).

To minimize E on the energy landscape, the network state is modified asynchronously from an initial state by updating each processing unit using the updating rule:

$$s_i = sgn\left(\sum_{j=1}^{N} t_{ij}s_j - I_i\right) \tag{3}$$

where the output of the s_i of the ith unit is fed to the input s_j of the jth unit asynchronously by the connection t_{ij} where there are N connections from other j processing units to the ith unit. The symmetry of the matrix t_{ij} with zero diagonal elements enables E to decrease monotonically with the updating rule.

In optimization problems, the concept is to associate the Liapunov function given in eqn (2) with the problem objective function by setting the connection weights and input biases appropriately.

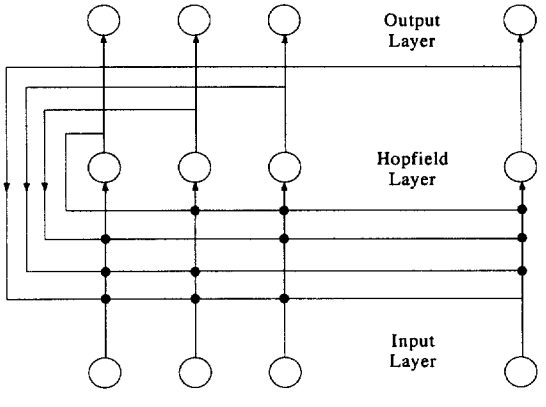

Fig. 5. A simple representation of a Hopfield network.

2.3.2 Simulated annealing

Simulated annealing is a probabilistic hill-climbing algorithm which attempts to search for the global minimum of the energy function. It carries out uphill moves in order to increase the probability of producing solutions with lower energy. The method carries out random search in order to find new configurations using the Boltzmann–Gibbs distribution:

$$P(\mathbf{S}) = \frac{e^{-\frac{E(\mathbf{S})}{T}}}{Z} \tag{4}$$

where T is the temperature of the system and Z is called the partition function which is of the form:

$$Z = \sum_{(\mathbf{S})} e^{-\frac{E(\mathbf{S})}{T}} \tag{5}$$

However, simulated annealing involves stochastic search for generating new configurations. In order to reach good solutions, a large number of configurations may have to be searched which involves slowly lowering the temperature and is, therefore, a very CPU time consuming process.

2.3.3 Statistical spin-glass models for neural networks

The simplest model of a physical system is the two-state Ising spin-glass model which allows the spin variables to take values of $+1$ or -1, or 0 or 1. A Potts model[23–26] is the generalization of the two-state Ising model which has Q equivalent ground states where all the spins are identical, but may take any one of the Q values.

The application of the two-state Ising model to partitioning of: graphs;[27,28] and adaptive finite element meshes is demonstrated in Ref.[19]. However, the present paper demonstrates the parallel implementation of the multi-state glass model which is used for the mesh partitioning problem. Thus, for a system of N neurons where each neuron can occupy Q discrete states and the state space of the neurons is restricted such that exactly one neuron at every Q state is allowed to be on. Using the binary values of 1 and 0 representing whether the state of a neuron is on or off, the neuron

restriction terminology may be written as:

$$\sum_{a}^{Q} s_{ia} = 1 \tag{6}$$

thus, a neuron i can be on (1) in only one of the Q possible states and off (0) in the remaining states.

Peterson and Söderberg[29] presented a new method for mapping optimization problems onto neural networks. By considering eqns (4) and (5), it has been shown[29,30] that the discrete sum of the partition function may be replaced by an integral over the continuous variables u_i and v_i and to evaluate (approximately) the integrand of this integral at the saddle point, therefore:

$$Z = C \prod_{j=1}^{N} \int_{-\infty}^{\infty} \mathrm{d}v_j \int_{-i\infty}^{i\infty} \mathrm{d}u_j \mathrm{e}^{-E'(\mathbf{V},\mathbf{U},T)} \tag{7}$$

where C is a constant and $\Pi\int\int$ refers to multiple integrals. For the two-state Ising model E' may be given in the form:

$$E'(\mathbf{V},\mathbf{U},T) = E(\mathbf{V})/T + \sum_{i=1}^{N} u_i v_i - \log(\cosh u_i) \tag{8}$$

and for the Q-state Potts model E' may be given in the form:

$$E'(\mathbf{V},\mathbf{U},T) = E(\mathbf{V})/T + \sum_{i=1}^{N} u_i v_i - \sum_{i=1}^{N} \log \sum_{k=1}^{Q} e^{u_{ik}} \tag{9}$$

and the multiple integrals may be determined using a saddle point expansion of E' which involves the mean field approximation that is found in Refs[28,29]. The saddle point positions for the Potts model are given by:

$$u_{ia} = \frac{-\partial E'}{\partial v_{ia}} \tag{10}$$

$$v_{ia} = \frac{\mathrm{e}^{u_i a}}{\sum_{b}^{Q} \mathrm{e}^{u_{ib}}} \tag{11}$$

where $\sum_a^Q v_{ia} = 1$

Further, the continuous variables, v_i, are used as approximations to the discrete variables at a given temperature (i.e. $v_i \approx < s_{ia} >_T$), thus the final value of v_{ia} approximates whether the value of s_i is 1 or 0.

Eqns (10) and (11) are the iterated asynchronously. This is based on updating the value of only one v_{ia} after each time-step ($t + \Delta t$).

Topping and Bahreininejad[20] demonstrated the use of neural networks in efficiently partitioning adaptive unstructured finite element meshes. The problem of partitioning meshes may be mapped onto a MFA Potts neural network and may be defined by an objective function in the form of the Hopfield energy function given in eqn (2). The Hamiltonian for the partitioning is, therefore, given by:

$$E(\mathbf{S}) = -\frac{1}{2} \sum_{i}^{N} \sum_{j}^{N} \sum_{a}^{Q} t_{ij} \mathbf{s}_{ia} \mathbf{s}_{ja} \tag{12}$$

where N is the number of elements and Q is the number of states which represents the number of partitions. This is carried out by assigning a binary unit of $s_{ia} = 1$ or $s_{ia} = 0$ to each element of the mesh defining which partition the element is to be assigned. The connectivity of the triangular elements is encoded in the t_{ij} matrix in the form of:

$$t_{ij} = \begin{cases} 1 & \text{if a pair of elements are connected by an edge} \\ 0 & \text{otherwise} \end{cases}$$

With this terminology, the minimization of eqn (12) will maximize the connectivity within partitions while minimizing the connectivity between the partitions. This has the effect of maximizing the boundaries and therefore using this term alone as the cost function forces all the elements to move into one partition, thus a penalty term is applied which measures the violation of the equal sized partition constraint.[28,29] Therefore, the neural network Liapunov function for the mesh partitioning problem is of the form of:

$$E(\mathbf{S}) = -\frac{1}{2} \sum_{i}^{N} \sum_{j}^{N} \sum_{a}^{Q} t_{ij} \mathbf{s}_i \mathbf{s}_j \pm \frac{\alpha}{2} \left(\sum_{i}^{N} \sum_{j}^{N} \sum_{a}^{Q} \mathbf{s}_{ia} \mathbf{s}_{ja} \right) \tag{13}$$

where a is the imbalance parameter which controls the balance between the partitions. The second term in eqn (13) ensures load balancing of the partitions and will be zero if the partitions are correctly balanced. From eqns (10) and (13), therefore:

$$u_i = \frac{\sum_{i}^{N} \sum_{j}^{N} \sum_{a}^{Q} (t_{ij} - \alpha) v_{ja}}{T} \tag{14}$$

The saddle pointeqns (11) and (14) is used to compute a new value of v_{ia} asynchronously. Initial values for the $\mathbf{V}$ matrix are assigned using (1)/(Q) plus a small noise factor increment. The temperature is lowered using a cooling factor (typically 0.95) after one complete iteration of eqns (11) and (14).

The Q spins associated with a particular element i can be looked upon as the probabilities of the element being assigned to each of the partitions and therefore from $\sum_a^Q v_{ia} = 1$, an element can occupy only one of the partitions.

The MFA Potts network was formulated for a parallel environment.[20] The parallel implementation of the annealing process was carried out using a pipeline network of transputers. The annealing process is applied using the asynchronous updating method. Fig. 6 shows the flowchart of the parallel algorithm.

First, the values of the initial parameters and the element information are sent from the master task to the worker tasks. The number of workers is equal to the number of required partitions. Next, the weight matrix t_{ij} which represents the elements connectivity is setup on all worker processors. The initial spin matrix, $\mathbf{V}$, is generated on the master and is distributed vectorially between the workers.

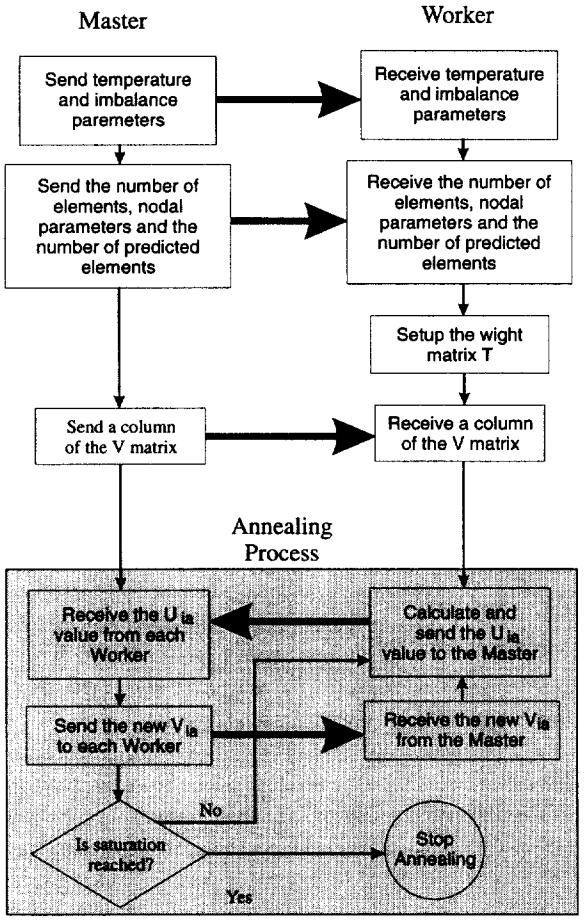

Fig. 6. Flowchart showing the parallel Potts annealing.

This means, for example, if the initial spin matrix is made of three columns representing three partitions, then each column is sent to a worker.

Once the annealing starts, each worker computes the value of eqn (14). Each worker then sends its result to the master where eqn (11) is computed and the resulting v_{ia}, values are sent back to the appropriate workers. This procedure is carried out until all the elements have been accounted for the annealing. The temperature is reduced on the master task. Once a desired level of saturation has reached (typically 0.999), the annealing stops. The saturation level is given by:

$$sat = \frac{(\sum v_{ia}^2)}{N} \tag{15}$$

Thus, at the end of annealing, the Master has all the optimized spin values in the form of a **V** matrix. This matrix represents the final partitioning solution.

The sequential MFA Potts neural network carried out the partitioning on a single transputer. The parallel code was run on a number of processors corresponding to the number of partitions required. To assess the efficiency of the parallel code, both the parallel and sequential codes were run for a total of 100 iterations on the problem domain shown in Fig. 7. The comparison for the 100 iterations are given in

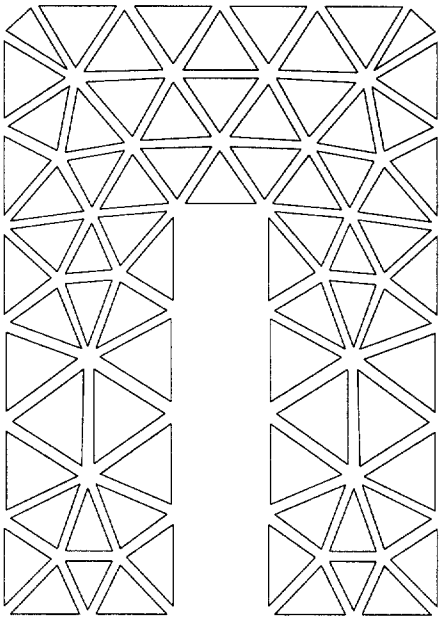

Fig. 7. The domain tested for the sequential-parallel MFA partitioning.

Table 1. Refs[19,20] provide more details on partitioning finite element meshes using MFA neural networks.

The initial coarse mesh shown in Fig. 7 was partitioned into seven partitions using the MFA Potts method. Fig. 8 shows the partitioned mesh based on the coarse mesh and Fig. 9 shows the partitioned mesh after re-meshing.

3 PARALLEL GENETIC ALGORITHMS

Genetic algorithms (GAs) are an interesting class of problem solving technique that combines the principles of population genetics and natural selection. These approximation algorithms have been successfully applied to several optimization problems which are difficult to solve by conventional mathematical programming.

GAs are stochastic methods that rely on a trade-off between exploitation of good solutions and exploration of space via genetic recombination based on probabilistic

Table 1. Comparison between sequential and parallel MFA Potts method run for a total of 100 iterations for partitioning of the mesh shown in Fig. 7

No. of partitions	Computation time (s)	
	Sequential code	Parallel code
1	6	6
2	13	8
3	20	9
4	27	11
5	33	12
6	40	13
7	47	15
8	54	17

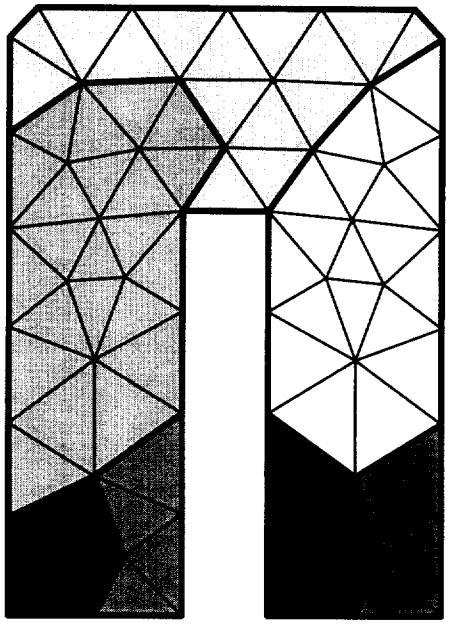

Fig. 8. The initial coarse mesh divided into seven partitions using the predictive MFA Potts neural network. The initial mesh has 75 elements.

control parameters. They are able to randomly sample large areas of the problem search space. Then, they evolve new search points based upon the performance of the old search points in the hope of improving the performance of the overall search.

The GA's search model attempts to mimic the genetic drift and Darwinian strife for survival. By manipulating symbolic representations of solutions, evolution in canonical GAs, is performed using three main operators: selection, crossover and mutation.

- Selection has the responsibility to impose selective pressure towards convergence exploiting good solutions according to the fitness rates of their objective function evaluations.
- Crossover, the most complete of the GA's operators, is able to evolve solutions through combination and inheritance of good features from different solutions. Crossover is basically responsible for local optimization using elements of direct search.
- Mutation promotes diversity at random and explores more global areas of the solution space providing a mechanism to escape from local optima.

Compared to conventional optimization techniques, GAs are population based instead of point-based, i.e. they attempt to evolve complex systems concurrently rather than to develop one and refine it. The implicit parallel properties gained by evolving a population of points in the search space concurrently[31,32] suggests that GAs have a natural mapping onto parallel architectures.

Although canonical GAs offer great potential for parallel implementations they also employ global information and a centralized control mechanism which constrains the parallelization to a certain level. Driven usually by the hardware availability, a variety of schemes for parallelizing GAs have recently been proposed. In an attempt to increase the degree of parallelization, reduce communication overheads and maximize efficiency of hardware resources, the majority of these parallel implementations introduce differences in structure[33–35] when compared to the canonical GA, especially in the areas of population, mating and generation models.

When significant modifications to the algorithm in the parallel model are required in order to take full advantage of the hardware available, it also seems reasonable to expect different behaviour on comparison with the original sequential algorithm. A variety of parallel algorithms have been previously proposed[36–38], whose models result in enhanced properties of population dynamics. Parallel genetic algorithms may be loosely classified according to three distinguished models: global, island and cellular.

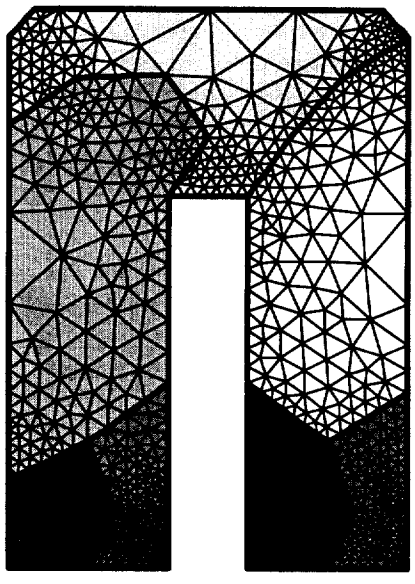

Fig. 9. The final remeshed subdomains divided into seven partitions using the predictive MFA Potts neural network. The final mesh has 1795 elements.

3.1 Global models

Serial GAs consider the total population as a single breeding (global) unit where mates are selected at random within the population (panmixia) and species evolve without any external influence. Hence, all individuals have to coexist in the same habitat and be equally available to each other for mating. Thereafter, reproduction is made in pairs of mates. This mechanism suggests that GAs inherit a certain degree of parallelism, since reproduction is an independent process and can be performed in a distributed manner. However, the overall control should reside in a single processor,

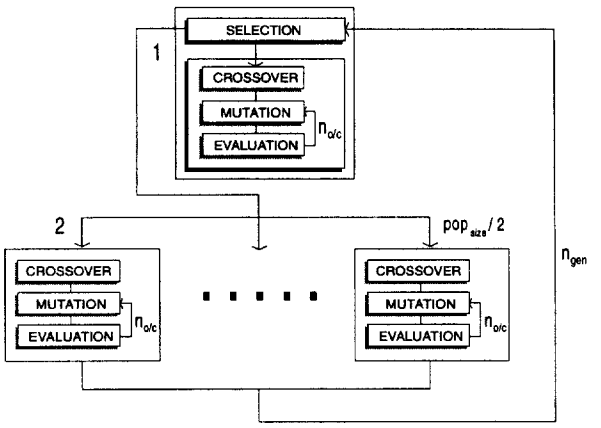

Fig. 10. Diagram of task distribution over network of n transputers using the population model of scheme A.

since information about fitness of the entire population must be available for the selection process. Hence, part of the algorithm has still to be processed in a sequential fashion.

If the GA is implemented in a traditional way, at each generation, the execution takes place on a single processor until selection is performed. Various schemes may be adopted to, thereafter, process crossovers, mutations and evaluations in a distributed manner.

However, the cost of generating and selecting a large population on a single processor is generally very high compared to the cost for reproduction of a pair of individuals. Therefore, parallelization in such a fashion is unlikely to be highly efficient unless the computational cost of the evaluations is very high.

Earlier implementations of these parallel genetic algorithms in shared memory machines (SMM) were examined by Grefenstette[39] in the early 1980s.

In scheme A, shown in Fig. 10, each processor including the root, is allocated two individuals for recombination. Thus, the maximum number of processors, $n_{\max}$ that may be utilized for a given population size is:

$$n_{\max} = \frac{1}{2} pop_{\text{size}} \tag{16}$$

If fewer than ($n_{\max}$) processors are available then more than one pair of individuals may be task farmed to each processor for recombination. For a perfect load balancing the population size should be equal to or a multiple of the number of processors used. In the serial algorithm used as model for parallelization in this study, the GEBENOPT (GEnetic Based ENgineering OPtimization Tool),[40] there is a possibility depending on the details *a priori* specified that more than two offspring are generated per pair of parents. In Fig. 10 this is shown by the loop on each processor generating $n_{\text{o/c}}$ offspring, where $n_{\text{o/c}}$ is the number of offspring per pair of parents.

Once the recombination has been performed by task farming the results, i.e. the best two offspring on each slave processor, are communicated back to the population on the root processor. Fig. 11 shows an alternative scheme referred to as scheme B, where again pairs of individuals are task farmed to the processors numbered 2 to $pop_{\text{size}}/2$. The crossover operation is performed on pairs of individuals by processors numbered 1 to $pop_{\text{size}}/2$ with each couple generating $n_{\text{o/c}}$ offspring which are distributed for mutation and fitness evaluation to a further ($n_{\text{o/c}} - 1$) processors. One individual is mutated and evaluated on each of the processors and the fitness tournament takes place at the next highest level. Hence, the maximum number of processors that may be utilized for a given population size is:

$$n_{\max} = \frac{1}{2} pop_{\text{size}} n_{\text{o/c}} \tag{17}$$

Since $n_{\text{o/c}}$ is always going to be two or greater then the maximum number of processors that may be utilized will be significantly greater for scheme B than for scheme A. A

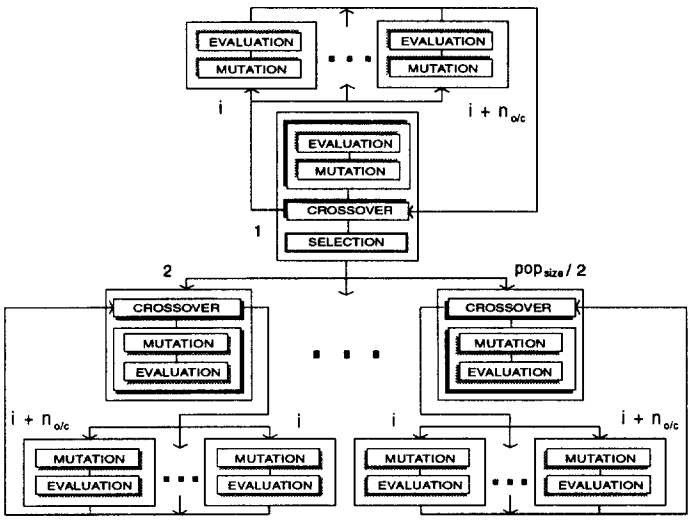

Fig. 11. Diagram of task distribution over network of n transputers using the population model of scheme B.

similar scheme was previously described in Ref.[41]; however, in that case only the fitness evaluations were performed in a distributed manner. For scheme B similar considerations apply concerning load balancing as discussed for scheme A.

Schemes A and B achieve higher efficiency when the fitness evaluation is computationally expensive compared to the generation of an individual. In this case, the communication overheads are also of very little significance, especially for small to medium sized populations. Previous implementations[41,42] of these schemes have shown nearly linear speedup. Both above schemes present no structural changes from the serial model, the only difference being the hardware accelerator. Therefore, no differences in behaviour have been observed with regard to the quality of solution and the same values of control parameters for the serial algorithm may also be adopted for the parallel one.

3.2 Island models

Although in theory this is not an impossibility, in nature's realm panmixia is only likely to occur in small and geographically isolated populations. In large populations, even though coexisting in the same continuous space, individuals tend to move and mate only within a surrounding region. This is a phenomenon known as isolation-by-distance.[43] Whether for segregated mating in neighbourhoods or for evolving semi or completely isolated sub-populations, the population is naturally subdivided.

In such a context, a parallel machine may be viewed as a global and complex environment composed of a number of independent regions. Each processor, then, corresponds to one isolated region or island and performs a complete GA on its own local population. These local or sub-populations may evolve in an isolated or semi-isolated regime but in any case the control and knowledge is local, i.e. the solutions or individuals are accessed and selected according to the statistics of the local population only. Hence, algorithms based in island models are coarse-grained parallel genetic algorithms (*cgp*GAs) whose parallelism comes from processing subsets of a serial population concurrently on a number of processors.

Another favourable aspect of this model is that by evolving small sub-populations the cost of some specialized operations may be substantially reduced. This is especially important for some advanced operators, present in some GAs, which may increase exponentially with the growth of the population. An example is when populations are ranked according to their fitness values. In this case, it is less expensive to rank and introduce individuals in small independent sub-populations than the entire population in a single rank.

This class of *p*GAs is suitable to solve large problems on a small network of processors. Increases in the number of processors enables the size of the sub-populations to be reduced down to a certain size, smaller than which the reproduction would be ineffective, resulting in incestuous crossbreeding and premature convergence. Thereafter, the use of a larger number of processors only results in a more representative sample which may lead to improvements in the quality of solution. This also may be viewed as an attractive feature since, in parallel GAs, the global population size may increase, but this will not necessarily introduce additional computational cost.

Thus, the efficiency of these algorithms may be directly related to the optimal serial population size. In problems where the optimal serial population size is already small, there is not much room for reductions in the population size and parallelization using such a model is unlikely to result in substantial speedup.

The simplest scheme of this model uses identical serial GAs to evolve complete isolated sub-populations in each processor. Isolated in the sense that there is no movement of individuals among local populations. The only communication between processes is to distribute the control parameters to all nodes and gather the final results to report to the user. Shonkwiler[44,45] presents some applications of selected problems using this *IIP* (independent and identical processing) model with isolated populations, which, he claims, achieves 'superlinear speedup' to convergence. However, this claim is supported by the philosophy that the GA, as a stochastic method, bases its achievements on a number of runs whose times have to be summed for the total time.

More refined schemes introduce a certain mobility of individuals towards adjacent or nearby sub-populations. The intermixtures among sub-populations attempt to increase diversity, thus reducing the problems due to lack of schemata for recombination and, hence, the avoidance of premature convergence.

Topping and Leite[46] discussed two new different diffusion schemes for structural engineering problems. In the first scheme, two individuals from different sub-populations travel at the same time for pairing. Hence, the reproductive population of a processor is composed only of mates from neighbour populations, i.e. without direct contribution of the

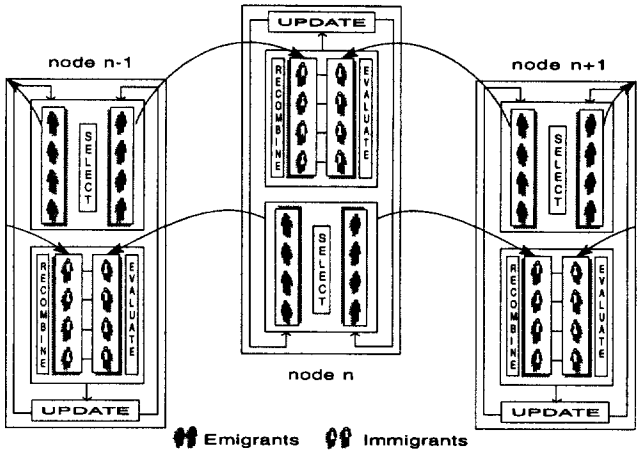

Fig. 12. Diagrma of task distribution over three generic subsequent nodes of a network of n processors in a ring topology using the sub-population model of scheme C.

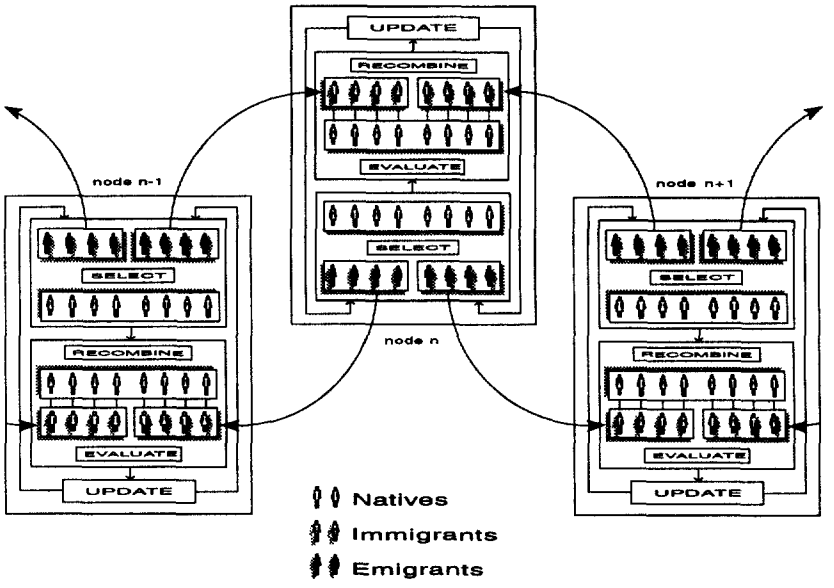

Fig. 13. Diagram of task distibution over three generic subsequent nodes of a network of n processors in a ring topology using the sub-population model of scheme D.

previous population. The diagram in Fig. 12 shows the distribution of tasks over three generic subsequent nodes of a ring topology of n processors using scheme C. In scheme D all recombinations are made with one mate from the resident population and another mate from the neighbouring population as shown in Fig. 13.

Once these sub-populations models address the problem of premature convergence, a remedial solution could be the hybridization as proposed by Mansour and Fox[47] and by M ühlembein *et al.*.[33] The use of hill-climbing techniques to refresh converged populations could minimize the likelihood of premature convergence.

3.3 Cellular models

The cellular models, are fine-grained parallel GAs (*fgp*GAs) composed of one large population subdivided in overlapping neighbourhoods, where a single individual is placed in each processor and each individual has its own particular neighbourhood. The parallel machine, in this case, may be understood as a body composed by a population of reproductive cells as shown in Fig. 14. If the body is structured in such a way that the cells have very limited mobility, reproduction only can occur between cells placed next to each other (neighbours). Hence, in these models each individual requests a mate from its neighbourhood for reproduction, based only on a local (neighbourhood) knowledge. After reproduction one offspring is chosen to replace one of its parents. The selection dynamics, then, are completely different from canonical GAs, where the two mates are equally selected from the entire population, and the two offspring may replace both parents.

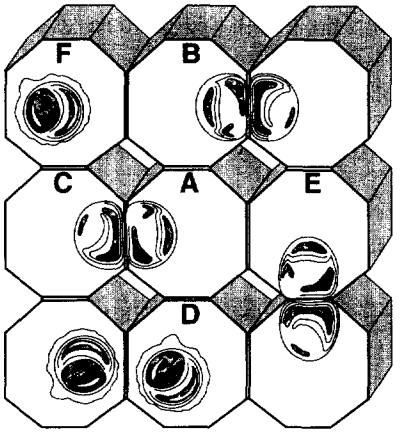

Fig. 14. Reproductive cells with restricted mobility, segregated by their environment.

As it also happens with *cgp*GAs, this class of algorithm finds its efficient applications among large problems (e.g. large populations), once the communication overheads are extremely reduced due to the small volume and short routes of messages. In fact, the high efficiency of the cellular models for some applications made researchers[48,49] affirm that these algorithms deliver 'superlinear speedup'. However, this impressive speedup is not achieved by the parallelization of the serial algorithm, but rather due to the better capabilities of the adapted parallel model.

An advantage of the *fgp*GAs is the capability to solve problems which combine very large strings (e.g. many variables and higher discretization) massive populations within reasonable computational time and without address memory problems. An inconvenience of these models is their lack of portability since they rely on complex topologies of processors and very expensive hardware.

3.4 Effect of communication on parallel GAs

The optimal sizes of parallel populations, sub-populations

and neighbourhoods are functions of the optimal serial population size and of the number of processors. Yet, the concept of optimal serial population size may be postulated as being the number of individuals required to produce sufficient schemata to solve the problem with minimum effort.

A direct distribution of a serial population over a number of processors allocates in general very few individuals to each local population. Then, if the sub-populations evolve without intercommunication, the lack of diversity induces the local populations to converge quickly, but to suboptimal solutions. Therefore, parallel models using totally isolated GAs can only afford little reduction in the population size. In order to allow the use of smaller populations, communication is required to promote intermixtures among segregated populations which will restore the diversity from time to time. These intermixtures among sub-populations may occur via migration of individuals or via diffusion of features.

The main claim behind models with segregated sub-populations is the occurrence of local and global optimization concurrently. Hence, by confining the search to different regions of the sample, the overlaps in the solution space may be reduced. In addition, since the competition is locally applied, different solutions may be evolved concurrently. A specific application for models using such population structure and communication mechanisms is in the optimization of multi-modal functions, in which the many local minima are of similar order of attraction to the global minimum. Competition, in canonical GAs, tends to eliminate many of the minima during the optimization process, which may eventually get stuck in a local minimum.

3.4.1 Migration

Migration was naturally introduced in the island models and the general idea was to send copies of individuals with high fitness to other processors, in order to be incorporated in their populations and make them available for selection. However, the introduction of new material may only be effective before a local population converges totally. Thereafter, a recessive genetic character will be immediately eliminated. Previous studies[50,51,38] have been carried out on a variety of schemes which adopt migration models. Although these migration schemes are based in this same premise, they are not necessarily equal. They may differ in the selection mechanisms, in the volume of communication (from a single individual to half sub-population) or in the broadcast connectivity (from two processors to the entire network). The number of communication connectivities is generally constrained by the topological capabilities of the hardware. The connectivity may also be static, maintained constant during the entire process or it may be dynamic, i.e. heuristically or randomly modified throughout generations.

A potential drawback of this type of communication is that in small and semi-isolated populations, external interference, i.e. communications among populations by migration, may affect the population evolution substantially. Relatively fit individuals injected in small populations may overtake the entire population within a few generations.

3.4.2 Diffusion

Diffusion is the interacting mechanism of the cellular models. This is a much more subtle process of intermixture among semi-isolated subsets of the population where individuals travel to visit other individuals in a small bounded area (neighbourhood) for mating. In Fig. 14, for example, since cell A only knows cells B, C, D or E, it shall mate with one of these four. Therefore, since there is no contact between cells A and F, features cannot be directly transferred from one to another. However, features may be transferred from cell A to cells B or C in a generation and from one of these two last to F in a subsequent generation.

A potential drawback is the formation of colonies of layouts sharing the same features and when combined within a colon it is unlikely to produce different individuals, since there is no difference in the genetic material.

Since interprocessor communication may be limited to four or six processors, it is not possible to physically connect all processors in the network. Thus, data sent from one processor to another may often pass through one or more intermediate processors which act as relay stations. The population structure of overlapping neighbourhoods limits the interprocessor communication to produce efficient implementations.

Later *cgp*GA implementations also use a neighbourhood structure to allow communications in two levels: migration for intermixtures within a neighbourhood and diffusion for intermixtures among neighbourhoods. This may be implemented using small neighbourhoods constituted of few local populations, thus, subsets of the local population moves to an adjacent sub-population (neighbour) for mating.

3.5 Parallel genetic algorithms in structural optimization

Optimization methods based on GAs have recently been applied, to various structural problems, and have demonstrated the potential to overcome many of the problems associated with gradient-based methods and mathematical programming techniques. The high computational cost of GAs, however, often limits their application to problems in which the design space can be made sufficiently small, even though GAs are most effective when the design space is large.

Although a GA-based structural optimization may require a large number of structural analysis, the optimization process is parallelizable to a high degree. For large structural optimization problems, distributed processing will enhance the efficiency of CAs by cutting the high computational cost which is the only drawback of the GAs.

In 1994, Punch *et al.*[52] showed the impact of parallelization in the design of laminated composite structures by genetic algorithms. These authors reported a superlinear speedup using an island model GA distributed over a cluster

of five SUN Sparc10 workstations connected via a local area network (LAN). In addition, the results obtained by the island model GA proved to be more refined than previous results obtained using the serial model. In the same year, Adeli and Cheng[53] used a global model *p*GA for the optimization of large structures such as high-rise building structures and space stations with several hundred members. In this adopted model only the evaluations were performed in a distributed manner. A maximum speedup of 7.7 times was achieved for a 35-story tower (with 1,262 elements and 936 degrees of freedom), using a shared memory machine (SMM) Cray Y-MP 8/864 with eight processors. In 1995, Adeli and Kumar[54] obtained again nearly linear speedup on the optimization of truss structures using the same global model, however in this case, distributed over a heterogeneous network of eleven IBM RS/6000 workstations. Later in the year, the same authors[55] compared the results obtained by this global model running in a distributed memory machine (DMM) with 512 processors, the CM-5, with the results previously obtained using the Cray-MP 8/864. The SMM showed a better performance than the DMM for this global model, using a same population size and number of processors. This difference in performance is expected since in SMMs the information exchange between processors is performed by means of global memory and, therefore, it does not introduce communication and synchronization overheads due to interprocessor communication. On the other hand, SMMs are, in general, provided with a small number of processors and their global memory are not able to hold data for very large problems. The use of a DMM with a larger number of processors allowed Adeli and Kumar[55] to solve very large problems such as a 147-story space tower structure consisting of 4,016 truss members and 817 nodes in an affordable computational time. Although these increase in hardware reduces the of efficiency of the system, it provides substantial acceleration to the execution process. In addition, DMMs enable the use of larger populations which may result in more representative sample and, consequently, in more refined results. Finally, the main deference in performance between high performance computing environments such as cluster of workstations and dedicated DMMs lies also on the communication process: dedicated DMMs use faster links for interprocessor communications. Yet, the heterogeneity of machines and the unpredictable multi-user occupation of networked workstation environment may introduce relative complexity in the load-balancing mechanism. Nevertheless, the cluster of workstations can in general provide a better price/ performance for structural optimization problems. Topping and Leite[46] examined the applicability and relative merits of the many parallel models and different parallel environments for engineering optimization. These authors employed a design problem of a cable-stayed bridge to illustrate the differences in performance on both speed and quality of solution of different parallel GA models. They discussed the different models and the computer architectures, topologies and applications where these models are expected to produce higher efficiency.

The results from these studies shown that, especially for large engineering problems, the parallel GAs perform better than the serial algorithm both in execution speed and quality of the solution.

3.6 Mesh partitioning using genetic algorithms and neural networks

In the first section of this paper, parallel and distributed computing were discussed. In the second and third sections parallel neural networks and genetic algorithms were discussed, respectively. Finally, in this section, a parallel version of a mesh partitioning technique for unstructured adaptive planar finite element meshes is described which utilizes a genetic algorithm and a neural network predictor. This method or partitioning is called the sub-domain generation method (SGM).

Conventional mesh-partitioning algorithms operate on the overall final mesh to obtain optimal partitions. With the SGM the overall mesh is not formed and, hence, a different strategy is adopted. The strategy used for the SGM relates to the adaptive mesh generation method described in Refs[2,56,57], where the adaptive refined mesh, which is the final mesh to be analysed, is generated using the following information:

- an initial coarse (background) mesh; and
- the element mesh parameters 6, for controlling the local mesh density.

In order to practically implement the SGM, the initial mesh must be partitioned into a suitable number of sub-domains such that each sub-domain of the refined mesh will include an approximately equal numbers of elements and the number of the interfacing boundary edges will be minimized. Advance knowledge regarding the number of elements in the final mesh is obtained by training a neural network to predict the number of elements that may be generated for each element from the initial coarse mesh.

The sub-domain generation method (SGM)[2,58] was originally implemented for planar convex finite element sub-domains using adaptive triangular unstructured meshes. The method comprises the following main components:

- a genetic algorithm-based optimization module; and
- a neural network-based redictive module.

The SGM was originally developed in sequential form for use on transputer arrays. Transputer-based systems are typical distributed memory MIMD architectures and usually have a central processor, or ROOT, which has one of its links connected to the HOST system. This processor is, therefore, responsible for the input–output support of the transputer system. When the idealization is of a large scale this task may overload the ROOT processor forming a bottle-neck in the analysis procedure. The memory and other performance considerations are detailed in Refs[2,58].

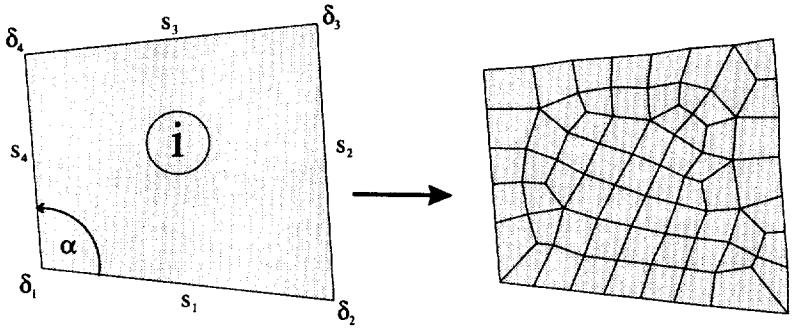

Fig. 15. The i-th element is considered to be an independent 'sub-domain' for remeshing.

The initial mesh is divided into two sub-domains by using a genetic algorithm regulated by an objective function which has a maximum value when the numbers of generated elements are equal in both the sub-domains and the number of interfacing edges is a minimum. This procedure is then applied recursively to each sub-domain.

The coarse initial mesh is, thus, partitioned into the desired number of sub-domains which are subsequently adoptively re-meshed. This re-meshing may be done concurrently, producing the final distributed mesh for use in a parallel finite element analysis. For generating sub-domain meshes in isolation to one another it is important that compatibility at the interfacing element edges is maintained. This problem may be easily solved on the basis of the technique used in Refs[2,58], where the nodal mesh parameter δ_n, was used instead of the element mesh parameter δ_e. The nodal mesh parameter δ_n may be readily calculated by nodal averaging.

Section 3.9 and Section 3.10 describe two approaches for the parallelization of the sequential SGM algorithm. First, the neural network predictor is described.

3.7 Neural networks

Artifical neural networks (ANNS) stem from neurobiological studies of the brain and the nervous system. In the SGM program, the backpropagation (BP) neural networks, as already described in Section 2.1, are used for approximation purposes. This type of network is based on the backpropagation algorithm[59] and its objective is to optimize a set of connecting weights so that a set of specified input patterns become associated with some output patterns. Once a desired set of weights has been achieved, the trained network may then be tested for some unknown input patterns for which an approximate output is given.

At the start of a training all the weights are set to random values and during the training the network continues adjusting the weights until a desired set of weights has been reached. This means that the training may be stopped once the difference between the desired output values of the training data and the actual output values of the network lies within a satisfactory range. The training may have little difficulty in reaching a desired state, but this may not be true in some cases of BP training where the training is difficult. The difficulty of a network in learning patterns may a result of the composition of the problem in terms of its proper and appropriate mapping for the neural network training or the poor data distribution.

Increasing the number of hidden units and/or increasing the number of hidden layers may contribute to a better learning; however, one should bare in mind that too many hidden units or layers may cause the network to memorize the training patterns (i.e. making perfect match), but be unable to generalize well and give good approximation for unknown patterns. Readers may find more information regarding the problems facing the BP training in Refs[60,61].

3.7.1 The BP neural networks for sub-domain generation

In Refs[2,58,62] a trained BP neural network was used as a predictive module for the decomposition of triangular finite element meshes. The purpose of the trained network is to approximate the number of elements which will be generated within each element of the coarse mesh when adaptive finite element re-meshing is carried out. This information is then used to partition this course mesh into several sub-domains using the genetic algorithm optimization module.

A similar approach has been carried out in Ref.[63] where a BP network was trained by using the data of several finite element meshes but with quadrilateral elements so that the trained network may then produce satisfactory approximations for the number of elements which may be generated in each of the coarse elements after re-meshing. The neural network software used was NETS 2.01[64] which is based on the classical BP algorithm and is in the public domain.

3.7.2 The training strategy and the network topology

As it was described in Section 3.6 the re-meshing of a coarse background element is done based solely on the basis of the element geometry and the nodal mesh parameters. Considering a mesh of quadrilateral elements only, Fig. 15 shows the re-meshing procedure for a single quadrilateral with all the parameters involved.[63] The element geometry is defined by the four side lengths of the element (s_1, s_2, s_3 and s_4) and the internal angle at the first corner, α. Thus, for training purposes a neural network with nine inputs (five for the geometric description and four nodal mesh parameters for each element) and one output (the number of elements

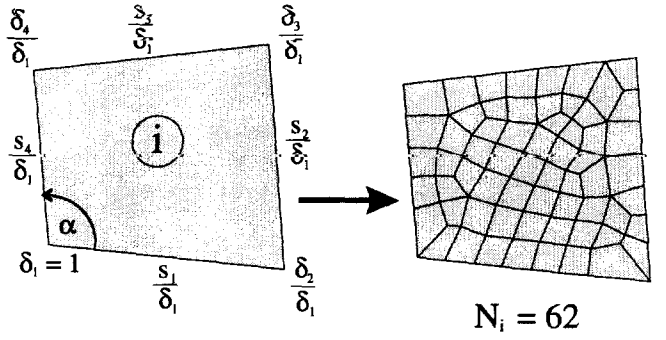

Fig. 16. Scaled representation of th input data.

generated) was foreseen. (The situation is similar when considering triangular finite elements and is described in Refs[2,58].)

However, knowing the fact that each nodal mesh parameter actually represents the size of the fine quadrilateral elements to be generated in the vicinity of the node, the scaling the four sides and the four nodal mesh parameters with one of the nodal mesh parameters would render that mesh parameter to be constant in the data set of the input stimuli. Fig. 16 shows this new representation of input data with the output number of elements N as opposed to the initial set in Fig. 15.

Hence, the neural network has eight inputs and one output. Besides the input and output layers the neural network has four hidden layers in a fully connected scheme as shown in Fig. 17.

3.7.3 The training data sets and performance test of the neural network

Again, the basic concept is described with regards to finite element meshes comprising quadrilateral elements. The equivalent description for triangular elements may be found in Refs[2,58].

The training data sets for the neural network were generated based on two simple background meshes shown in

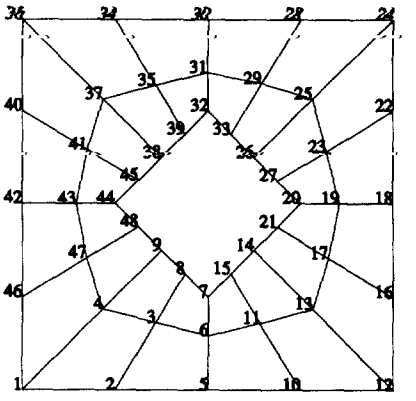

Fig. 18. The first mesh used for training the NN (comprising 32 elements).

Figs 18 and 19 comprising 32 and 28 elements, respectively. The nodes 1 and 12 were fixed in both directions in the mesh shown in Fig. 18 and nodes 1 and 24 were fixed in both directions in the mesh shown in Fig. 19.

Very simple loading conditions were imposed on these training meshes. One single concentrated load was applied in either the horizontal direction P_x or in the vertical direction P_y with different magnitudes but always one at a time at certain nodes of the idealization selected randomly. Table 2 gives the exact choice of loaded nodes. Thus, the first and the second training mesh were subjected to 36 and 22 simple load cases, respectively.

For each of these load cases the meshes were analysed, then finite element error analysis was carried out, then the element mesh parameters were determined. By nodally averaging these parameters the nodal mesh parameters were calculated and the meshes were adoptively re-meshed using these parameters by the parallel quadrilateral mesh generator.[57] The results of these re-meshings were saved in a file in such a way that it contained the following information for each background element in the training meshes for each load cases: the geometric definition of the element and the nodal mesh parameters in the scaled form described

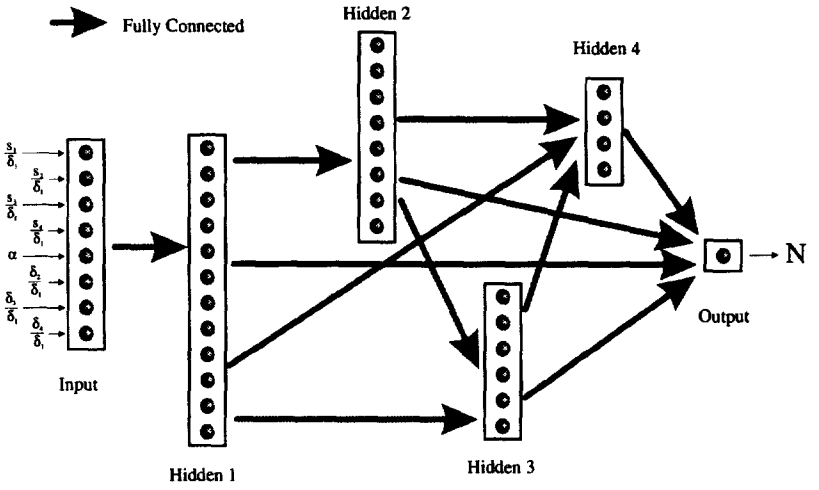

Fig. 17. The six-layer fully connected network used for quadrilateral elements.

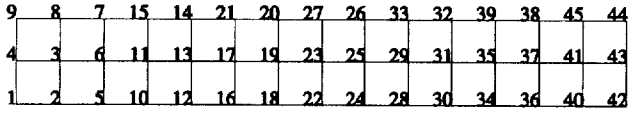

Fig. 19. The second mesh used for training the NN (comprising 28 elements).

Table 2. The loading cases for the training meshes

Load type	Magn.	Loaded nodes	No. load cases
First training mesh in Fig. 18			
P_x	100	16, 18, 20, 22, 28, 30, 32, 36, 40, 44	10
P_x	200	16, 18, 20, 22, 28, 30, 32, 36, 40, 44	10
P_y	100	4, 5, 7, 19, 20, 28, 30, 32	8
P_y	− 100	4, 5, 7, 19, 20, 28, 30, 32	8
Total number of load cases for this mesh: 36			
Second training mesh in Fig. 19			
P_x	100	4, 5, 6, 7, 15, 21, 33, 38, 44	9
P_y	200	4, 5, 6, 7, 15, 21, 33, 38, 44	9
P_y	− 100	12,39,42,44	4
Total number of load cases for this mesh: 22			

in the previous section (eight values) and the actual number of refined elements generated in that coarse element (one value).

Thus, the first training mesh gives $32 \times 36 = 1152$ sets of data. Similarly, the second mesh gives $28 \times 22 = 616$ sets of training data. This altogether would result in $1152 + 616 = 1768$ data sets, but some of the data were too close, in particular when the generated number of elements is equal to 1, so these would not affect the training, and only increase the computational load. For this reason the training data set was pruned to enhance the training efficiency by removing 254 sets from the data file, leaving 1514 data sets for the training.

The neural network was trained with respect to limiting the RMS error values of the training meshes. The RMS error is an overall error measure for a mesh based upon the difference between the predicted (p_k) and the actual (a_k) number of elements generated for each background element:

$$\mathrm{RMS}_{\mathrm{error}} = \sqrt{\frac{\sum_{k=1}^{N_i} (a_k - p_k)^2}{N_i}} \tag{18}$$

where N_j is the number of coarse elements in that mesh.

Table 3. The training parameters

Training parameters for the NETS 2.01	
Max. weight	1.0
Min. weight	− 1.0
Learn rate	0.25
Momentum	0.50
RMS_{error}	0.0065
No. of cycles	3000

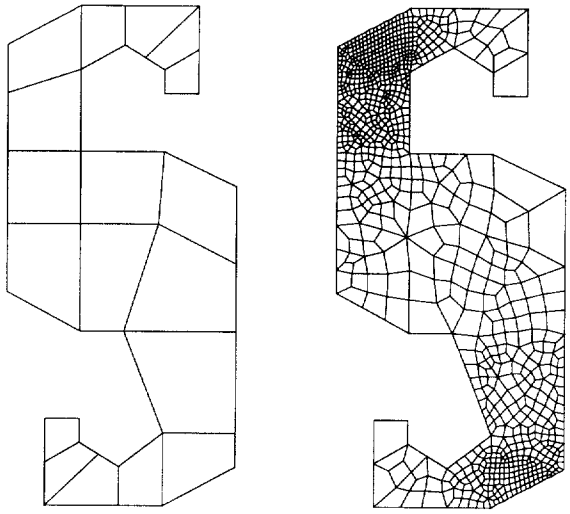

Fig. 20. The first mesh ('Hook') used for testing the NN (comprising 18 elements) and its re-meshed version (comprising 909 elements).

The network was trained using the parameters in Table 3, until the desired accuracy was achieved. Then three new meshes were used to test the performance of the neural network. These meshes and their re-meshed versions are shown in Figs 20–22. The RMS error values achieved the trained network are displayed in Table 4.

This network gives enhanced accuracy over the one which was used in the original program of SGM[58] developed for meshes built from triangular finite elements. Its main virtue is the ability to estimate the generation of up to 600 elements per coarse background element with good accuracy.

The training of the neural net could be undertaken in parallel as described in Section 2.1 and Ref.[16].

3.7.4 Building the neural network into the SGM program
After the training reached the required error tolerance (0.0065 RMS), 'flash code' was created by the neural network software. This is a piece of source code written in C which can be built into any other software in a form of a function call. This code contains the final trained values of

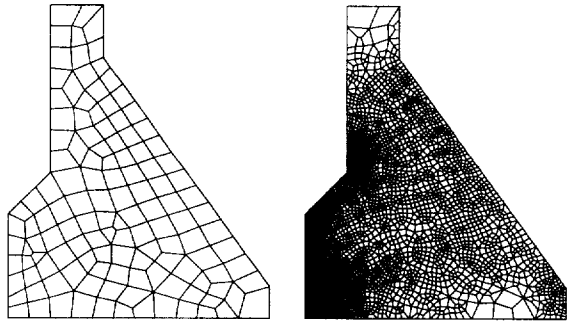

Fig. 21. The second mesh ('dam') used for testing the NN (comprising 127 elements) and its re-meshed version (comprising 5390 elements).

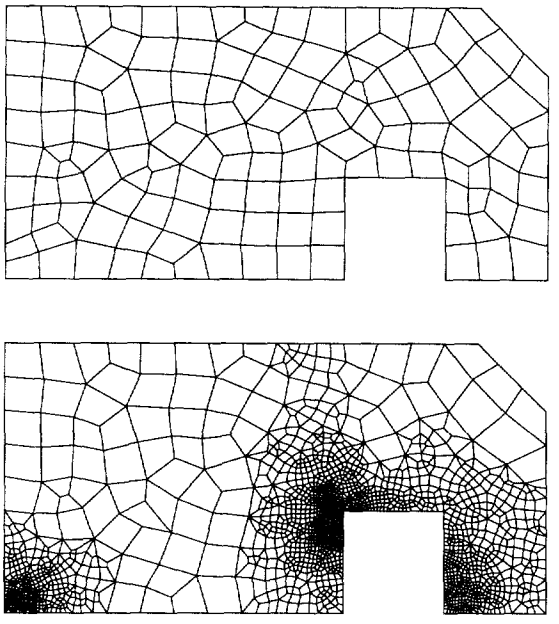

Fig. 22. The third mesh ('ex61') used for testing the NN (comprising 137 elements) and its re-meshed version (comprising 1999 elements).

the weights for the neurons, and applies them to the processing of each new set of input data. It is clear from Section 3.7.2, Section 3.7.2 and Section 3.7.3 describing the topology and the training of the neural network used for sub-domain generation that there are two separate networks for meshes comprising triangular and quadrilateral elements. They are both built into the SGM program as separate C functions and called as necessary.

In the case of the quadrilateral version this C function has eight input parameters which are the input values for the neural network and a ninth parameter which is the output value, the number of elements to be generated in a background element (as described in Ref. Fig. 17). This function is called at the beginning of the program for each background element and the output values are stored in a vector variable. This vector is then subsequently used many times over in the genetic algorithm module for the evaluation of the fitness function. For every individual, i.e. every new position of the divider vector **D**, the coarse elements are split into two groups by **D**. Finally, for both of the groups the total number of elements is approximated after re-meshing based on the neural network output values:

$$NE_{f1} = \sum_{k=1}^{NE_{i1}} N_k \text{ and } NE_{f2} = \sum_{k=1}^{NE_{i2}} N_k \tag{19}$$

Table 4. The test results of the trained net

Test mesh	'hook'	'dam'	'ex61'
No. Elements in coarse mesh	18	127	137
No. Elements in refined mesh	909	5390	1999
RMS_{error} values	7.795	4.918	2.789

where NE_{i1} and NE_{i2} are the number of coarse background elements on the first and second parts of the initial mesh after the splitting. The values of NE_{f1} and NE_{f2} are substituted into the fitness function evaluation given later.

3.8 Genetic algorithm for mesh partitioning

3.8.1 Cutting the mesh

The SGM was developed for planar convex finite element meshes. The mesh had to be planar and convex because of the algorithm used to bisect the mesh, which works on the following manner: the finite element mesh is cut by a dividing vector **D** in its plane as shown on Fig. 23. This vector is defined by three variables:

$$\mathbf{D} = \{x, y, \theta\}^T \tag{20}$$

where:

$$x \in [x_{\min}, x_{\max}] \tag{21}$$

$$y \in [y_{\min}, y_{\max}]$$

$$\theta \in [0, \pi]$$

The elements of the finite element domain are then divided into two sets by virtue of their centroidal locations, i.e. either to the left or to the right side of the generated vector.

The optimal position of this dividing vector **D** has to be determined in the optimization procedure of the domain decomposition accordingly to the conditions described in Section 3.6. Thus, these three parameters (x,y and θ) become the design variables for the genetic algorithm based optimization module.

3.8.2 Objective function—fitness function

The optimum criteria is to have an equal number of elements in both the resulting sub-domains whilst the number of interfacing nodes should be at a minimum. This is achieved by an objective function constructed in the following way:

$$g(x, y, \theta) = |NE_{f1} - NE_{f2}| + C_{cf} \tag{22}$$

where NE_{f1} and NE_{f2} are the number of finite elements in the first and second half of the mesh, respectively, after

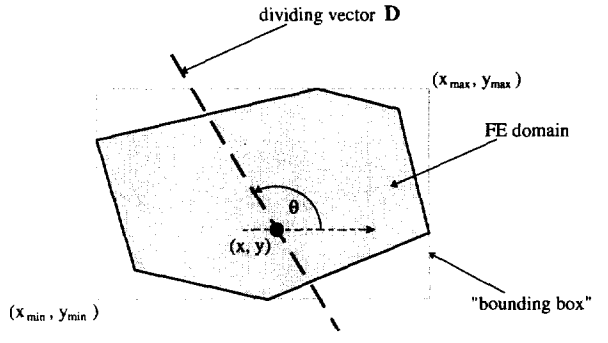

Fig. 23. Vector **D** generated to bisect the FE domain.

being cut with the current dividing vector **D**. Thus, if NE_f would represent the number of elements in the full mesh then $NE_f = NE_{f1} + NE_{f2}$. The determination of NE_{f1} and NE_{f2} are described in Section 3.7.4 by eqn (19).

The calculation of the cumulative square root value C_{cf} of the number of interfacing edges C_f in the bisected mesh is detailed for meshes comprising triangular elements in Refs[2,58] and for quadrilateral elements in Ref.[63].

The genetic algorithm procedure tends to maximize the fitness of its individuals. The function in eqn (22) is mapped into a fitness function $f()$ so that a maximum $f()$ corresponds to a better cut. This is accomplished using the following expression:

$$f(x, y, \theta) = C_{\max} - g(x, y, \theta) \tag{23}$$

where $C_{\max}$ is a suitably chosen large constant number, selected as NE_f in the SGM implementation.

3.8.3 Convergence criteria

By giving the genetic algorithm a large sample space and allowing it to process a large number of generations, one may say with a certain degree of confidence that the solution reached is either the global optimum or quite close to it. To prevent the genetic algorithm from processing too large a number of generations which would render the mesh-partitioning computationally expensive the following convergence criterion was employed. For the population size of 50, chromosome length of 3×10 bits, crossover probability of 0.8 and mutation probability of 0.0333, the genetic algorithm procedure was terminated if no improvement over the best design (fittest individual) was made within two consecutive generations. In addition to this criterion, a maximum limit of 500 was set on the number of generations, but this criterion was never invoked during the test runs of the SGM.

3.9 Recursive bisections on the ROOT with a parallel GA

The first approach to parallelize the SGM method is relatively simple and involves the parallelization of the genetic algorithm part only. This is the part which is the most time consuming in the algorithm of the SGM. The logical structure of this parallelization is represented in Fig. 24.

This algorithm is in reality a task farming[2] parallelization scheme hence it maintains separate sub-populations on each worker processor and they are communicated to and compared on the ROOT. This corresponds to an island model as previously discussed. There is no inter-processor communication between these workers. This procedure is then called recursively for each sub-domain, until the desired number of splits have been achieved.

On completion of such domain decomposition the parallel mesh generator[2,56,57] may be invoked to refine the meshes of each sub-domains. This mesh generator uses the same type of task farming parallelization scheme.

The disadvantage of this form of parallelization of the SGM is that although the mesh is decomposed using the coarse mesh, in practice the ROOT processor will have to handle all the information for the refined final mesh. In addition, the results of the mesh generation have to be redistributed over the processor network for analysis.

3.10 Parallel recursive bisections in a tree structure

In this algorithm each processor uses a local SGM subroutine to decompose the mesh, then sends one of the two resulting sub-domains on to another processor, as shown in Fig. 25, which will be still idle. After the completion of the sub-domain generation each processor is equipped with part of the initial mesh for adaptive re-meshing and processing during the explicit finite element analysis.

These meshes must be first refined with the adaptive mesh generator, but this mesh generator will generally be sequential or parallel on a single processor. The considerations regarding the newly generated boundary nodes mentioned in Section 3.6 still apply with the addition that the lack of global node numbering scheme[2,65] requires a special strategy for the synchronization of these nodes. The worker processors must keep track of the constantly changing new positions of their boundary nodes due to the progressing decomposition. This is essential for the parallel finite element analysis routine, which may commence

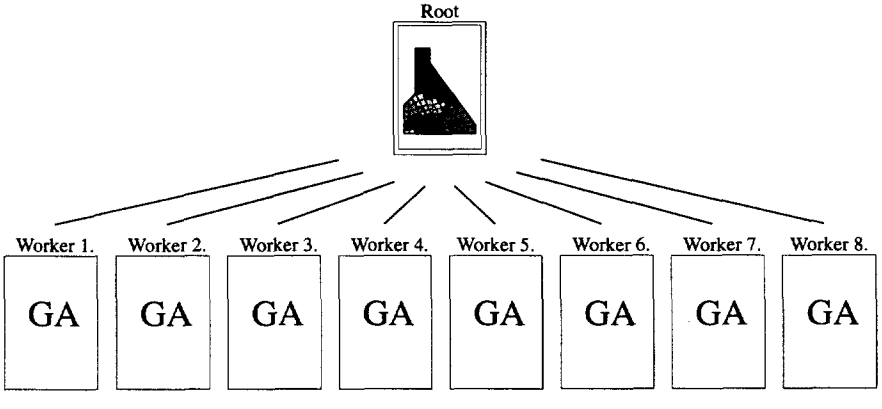

Fig. 24. Task farming approach to the parallelization of the SGM.

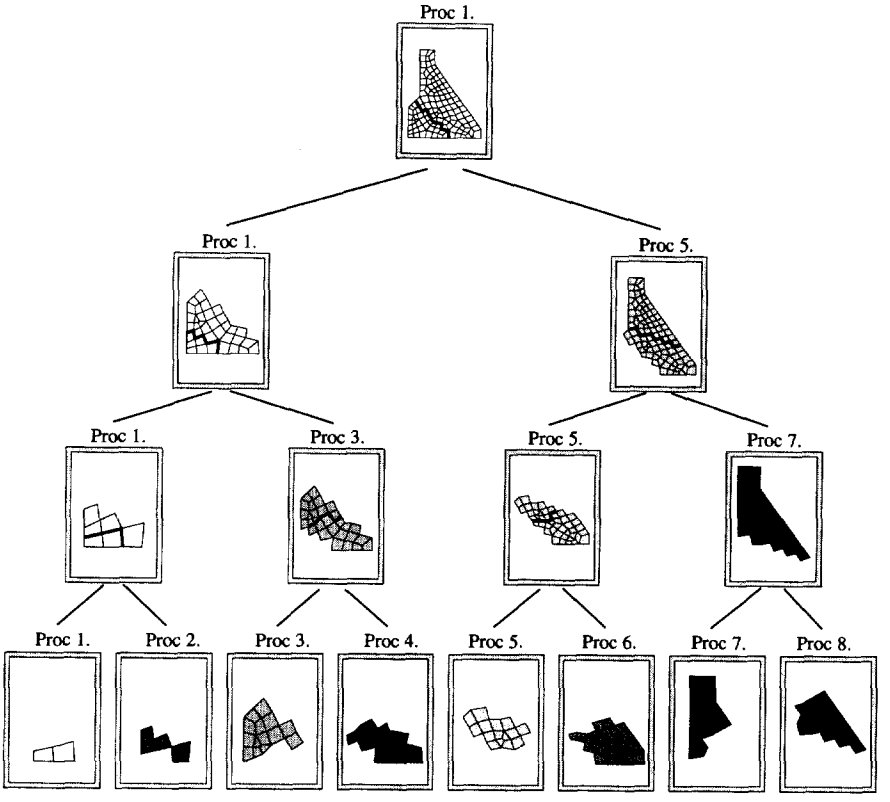

Fig. 25. Recursive tree structure for the parallelization of the SGM.

immediately after the completion of the last split on each of the processors and the subsequent re-meshing.

Despite these administrative difficulties, this second scheme maintains the initial virtue of the SGM, i.e. to handle the final mesh information in a totally distributed sense. The description of the final refined mesh never goes through the ROOT processor, apart from the final output of the finite element analysis at the very end of the analysis procedure.

3.10.1 Task connectivity scheme

This tree structure approach for the parallelization of the SGM is slightly more complex than the parallel genetic algorithm technique as it involves a development of a special routing scheme to deliver the description of the partitions to the idle processors as the sub-domain generation proceeds in a logical tree structure. This routing scheme, shown in Fig. 26, has to conform to the routing algorithm[2,66] used by the finite element program which will be based on an explicit time-marching finite element algorithm.

The master task loads down the coarse finite element mesh description from the HOST, but the analysis is carried out by identical worker tasks. During the explicit finite element analysis, these worker tasks have to exchange data relating to the common interfacing finite element nodes at each iteration step. This requires the possibility of direct communication between every worker task because of the arbitrary shape and possible connection layout of the sub-domains. To make this communication as efficient as possible a special routing scheme was developed in Refs[2,66] which utilizes all the four physical (hardware) links of a transputer, but rendering this hardware specific structure invisible for the application programs.

The structure of this connectivity scheme is best described in relation to Fig. 26. As it is shown, the components of the application program, i.e. the master task and the worker tasks, have as many communication channels as many worker tasks are in the system. (That is eight in this case.) If one of these tasks is about to communicate with any other it sends a message along the relevant channel from its set, as if the channel was connected to the appropriate task. This is not always physically possible because the transputer has only four physical links and each of them could only be used to carry one (hardware link) channel.

The router task resolves this problem by handling the processor's limited number of physical links as they are connected to each other using these hardware link channels and in doing so it logically creates a virtual channel system permitting general processor to processor communication. An application module has all its channels connected to the appropriate router task, running on the same processor, but these channels are internal 'on-processor' channels. The router task monitors all these channels and on the receipt of the message it adds a very small address header to it. The address headers are used by only the router tasks and passed on to the next router on a neighbouring processor.

The router task knows the final destination of the message from the ID number of the internal channel through which it

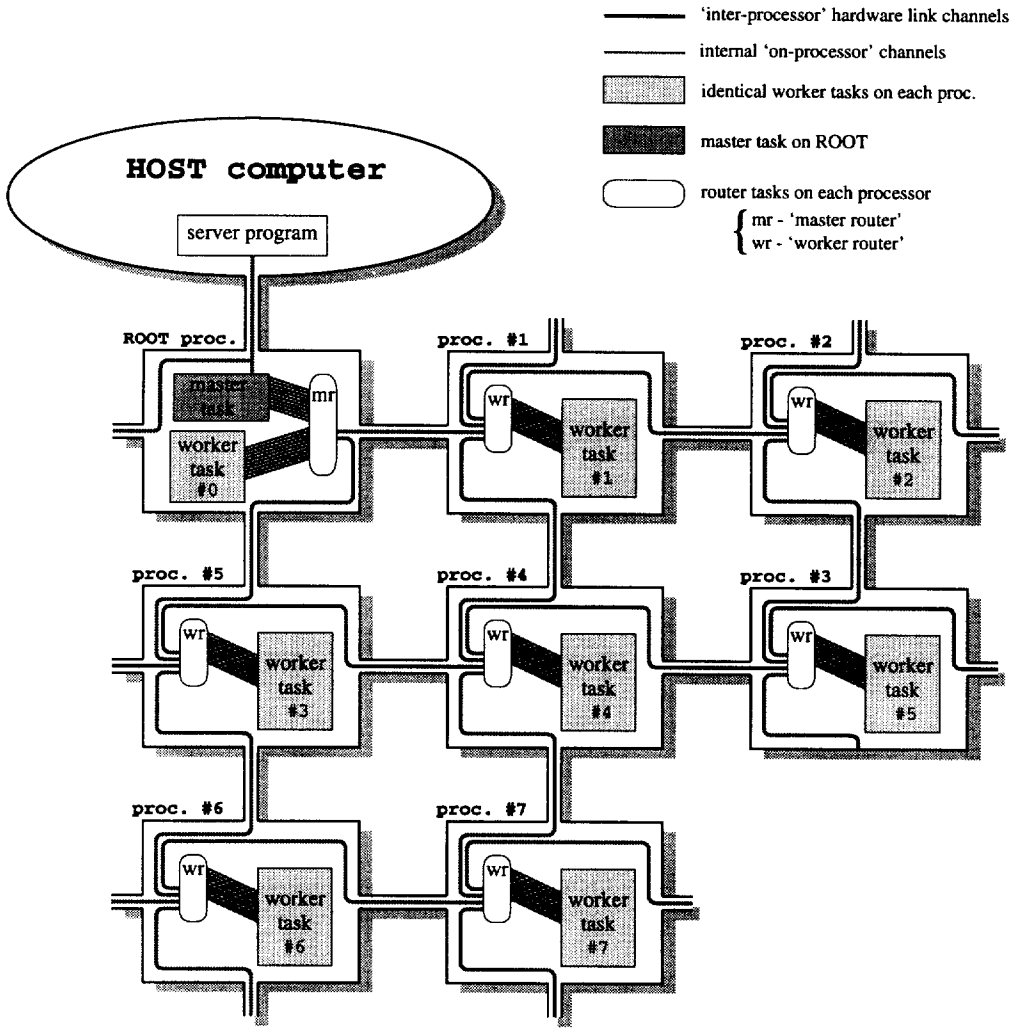

Fig. 26. Transputer routing scheme for parallel finite element analysis.

received the message from the application task. When the message finally arrives at the target processor, the router task decodes it (i.e. removes the header) and passes the message to the application program via the appropriate channel.

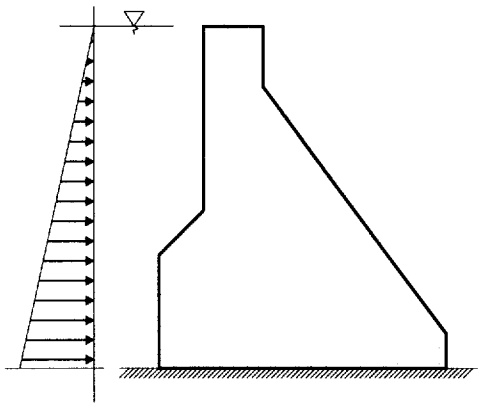

Fig. 27. SGM example: domain with a cross sectional shape of a dam.

3.11 Example

A domain with the shape of a cross section of a dam, shown in Fig. 27 with an in-plane horizontal distributed load on the left hand side and the bottom nodes restrained, was uniformly meshed and the initial mesh comprised 127 elements, as shown in Fig. 28.

The parallel implementation of the SGM was applied to this initial mesh and the mesh was divided into eight sub-domains by performing three recursive bisections, as shown in Fig. 28. The sub-domains obtained were independently re-meshed and the re-meshed sub-domains are shown in Fig. 29. The resulting final mesh comprises 2730 elements.

Here, the nodes along the boundaries of the sub-domains were fixed and nodal smoothing was undertaken on the sub-domains distributed among the processors. It would be possible to smooth the nodes on boundaries, but this would require inter-processor communication.

Table 5 shows the timing results of parallel SGM for this example. Table 6 shows the number of elements generated

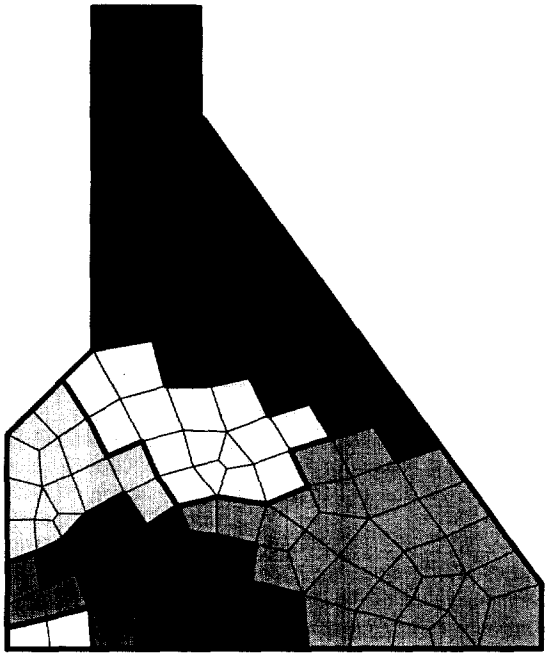

Fig. 28. SGM example: the initial mesh (127 elements) divided into eight sub-domains.

in each sub-domain, and the number of cut interfaces after the re-meshing of these sub-domains.

Table 7 shows the execution times of PSGM on different number of processors and the notional speed-up which can be calculated.

The row 'Max. no. of processors used' in Table 7 refers to the parallel version of SGM running on multiple processors. This is the maximum number of processors being used at the end of the procedure, when the progressing tree structure reached its lowest level.

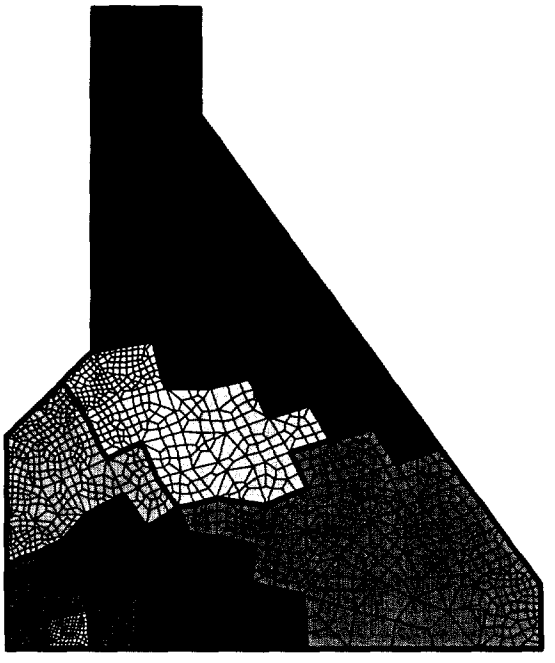

Fig. 29. SGM example: the re-meshed sub-domains (2730 elements).

Table 5. PSGM example: timing details for the individual splits

Timing for:	Rec. 1	Rec. 2	Rec. 3	Total time
Proc. 1	138.44 s	6.72 s	1.70 s	146.86 s
Proc. 3			9.80 s	154.96 s
Proc. 5		77.48 s	10.51 s	226.43 s
Proc. 7			19.23 s	235.15 s
Max. value	138.44 s	77.48 s	19.23 s	235.15 s
Total execution time:				237.20 s

Table 6. PSGM example: comparison of the actual number of generated elements per sub-domain versus the ideal number of elements that should have been generated using the PSGM

Sub-domain No.	Generated elements (actual)	Generated elements (required)	Difference	Percentage difference
1	341	341.25	− 0.3	− 0.07
2	344	341.25	2.8	0.81
3	339	341.25	− 2.3	− 0.66
4	338	341.25	− 3.3	− 0.95
5	337	341.25	− 4.2	− 1.25
6	335	341.25	− 6.2	− 1.83
7	372	341.25	30.8	9.01
8	324	341.25	− 17.2	− 5.05
$\sum_{i=1}^{n} \lvert x_i \rvert$	2730	2730	67.0	19.63
Total number of cut interfaces: 212				

Table 7. Comparison of the execution times and the speed-up

Parallel SGM on a single proc.	277 s
Parallel SGM on multiple proc.	237 s
No. sub-domains generated	8
Max. no. of processors used	4
Estimated sequential time	277s
Notional speed-up	1.17

Although the speed-ups for the partitioning are not particularly high the parallel version of the SGM has the inherent advantage that the sub-domains are delivered to the distributed processors without the requirement to form the complete finite element model on the root processor.

4 CONCLUDING REMARKS

In this paper, aspects of: parallel and distributed computing; parallel neural networks; and parallel genetic algorithms have been discussed. The use of these techniques for domain decomposition of unstructured adaptive finite element meshes is described.

ACKNOWLEDGEMENTS

The research described in this paper was made possible by the support afforded by HCM contract no: CHRX-CT93-0390 (DG12COMA) 'Advanced Finite Element Solution

Techniques on Innovative Computer Architectures'.The research described in this paper was supported by Marine Technology Directorate Ltd research contracts: 'High Performance Computing for Marine Technology Research', (ref: GR/J22191); and 'High Performance Adaptive Finite Element Computations for CAD of Offshore Structures using Parallel and Heterogeneous Systems', (ref: GR/J54017). The research described was also supported by the Systems Architecture Committee of the U.K. Engineering and Physical Sciences Research Council through the research contract: 'Domain Decomposition Methods for Parallel Finite Element Analysis', (ref: GR/J51634).The authors thank other members of the Structural Engineering Computational Technology (SECT) Research Group in the Department of Mechanical and Chemical Engineering, Heriot-Watt University for useful discussions.

REFERENCES

1. Topping, B. H. V. and Szlveri, J., Parallel processing, neural networks and genetic algorithms. In *Proceedings of the 2nd International Conference Computer Applications Research and Practice*, Vol. 3, University of Bahrain, April, 1996.
2. Topping, B. H. V. and Khan, A. I., *Parallel Finite Element Computations*. Saxe-Coburg Publications, Edinburgh, U.K., 1996.
3. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V., *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, MA, 1994.
4. Gropp, W., Lusk, E. and Skjellum, A., *Using MPI—Portable Parallel Programming with the Message-Passing System*. MIT Press, Cambridge, MA, 1995.
5. Hebb, D. O., *The Organization of Behavior*. Wiley, New York, 1949.
6. Jorgensen, C. C., Neural network representation of sensor graphs in autonomous robot path planning. In *IEEE 1st International Conference on Neural Networks IV*, San Diego, U.S.A., June 1987, pp. 507–515.
7. Van Der Smagt, P. and Kröse, B. J. A., A real time learning neural robot controller. In *Proceedings of the 1991 International Conference on Artificial Neural Networks*, Espoo, Finland, June 1991, pp. 351–356.
8. Miller III, W. T., Real time application of neural networks for sensor-based control of robot with vision. *IEEE Transactions, Systems, Man and Cybernetics*, 1989, **19**, 825–831.
9. Fukushima, K., Neocognitron: a hierarchical neural network capable of visual pattern recognition. *Neural Networks*, 1988, **1**, 119–130.
10. C. Mead, *Analog VLSI and Neural Systems*. Addison-Wesley, Reading, MA, 1989.
11. Marr, D., *Vision*. W. H. Freeman, San Francisco, U.S.A., 1982.
12. Izui, Y. and Pentland, A., Speeding up back propagation, in *Theory Track Neural and Cognitive Sciences Track of the Proceedings of the International Joint Conference on Neural Networks*, Vol. 1, ed. Caudill, M., IJCNN-90-WASH-DC. Lawrence Erlbaum, NJ, 1990, pp. 639–642.
13. Hagiwara, M., Accelerated back propagation using unlearning based on Hebb rule, in *Theory Track Neural and Cognitive Sciences Track of the Proceedings of the International Joint Conference on Neural Networks*, Vol. 15, ed. Caudill, M., IJCNN-90WASH-DC. Lawrence Erlbaum, NJ, 1990, pp. 617–620.
14. Cho, S.-B. and Kim, J. H., An accelerated learning method with backpropagation. In *Theory Track Neural and Cognitive Sciences Track of the Proceedings of the International Joint Conference on Neural Networks*, Vol. 1, ed. Caudill, M., IJCNN-90-WASHDC. Lawrence Erlbaum, NJ, 1990, pp. 605–608.
15. Rumelhart, D. E., Hinton, G. E. and Williams, R. J., Learning internal representation by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, eds. Rumelhart, D. E. & McClelland, J. L. Foundations, MIT Press, 1986.
16. Khan, A. I., Topping, B. H. V. and Bahreininejad, A., Parallel training of neural networks for finite element mesh generation. In *Neural Networks and Combinatorial Optimization in Civil and Structural Engineering*, eds. Topping, B. H. V. & Khan, A. I. Civil-Comp Press, Edinburgh, U.K., 1993, pp. 86–94.
17. Parker, K. L. and Thornbrugh, A. L., Parallelized BP training and its effectiveness. In *Theory Track Neural and Cognitive Sciences Track of the Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, ed. Caudill, M., IJCNN90-WASH-DC. Lawrence Erlbaum, NJ, 1990, pp. 179–182.
18. Peterson, C. and Rognvaldsson, T., An introduction to artificial neural networks. In *Proceedings of 1991 CERN School of Computing*, Vol. 92, No. 2, ed. Verkerk, C., CERN, May 1992, pp. 113–170.
19. Bahreininejad, A., Topping, B. H. V. and Khan, A. I., Subdomain generation using multiple neural networks models. Iin *Parallel and Vector Processing for Structural Mechanics*, eds. Topping, B. H. V. & Papadrakakis, M. Civil-Comp Press, Edinburgh, 1994.
20. Topping, B. H. V. and Bahreininejad, A., Subdomain generation using parallel Q-state Potts neural networks multiple neural networks models. In *Developments in Neural Networks and Evolutionary Computing for Civil and Structural Engineering*, ed. Topping, B. H. V. Civil-Comp Press, Edinburgh, 1995, pp. 65–78.
21. Hopfield, J. J. and Tank, D. W., Neural computation of decisions in optimization problems. *Biological Cybernetics*, 1985, **52**, 141–152.
22. Kirkpatrick, S., Gelatt, C. D. Jr and Vecchi, M. P., Optimization by simulated annealing. *Science*, 1983, **220**(4598), 671–680.
23. Wu, F. Y., The Potts model. *Reviews of Modern Physics*, 1982, **54**(1), 235–268.
24. Fischer, K. H. and Hertz, J. A., *Spin Glasses*. Cambridge University Press, U.K., 1991.
25. Yeomans, J. M., *Statistical Mechanics of Phase Transition*. Oxford University Press, U.S.A., 1992.
26. Hertz, J., Krogh, A. and Palmer, R. G., *Introduction to the Theory of Neural Computing*. Addison-Wesley, U.S.A., 1991.
27. Bilbro, G., Mann, R., Miller III, T. K., Snyder, W. E., Van den Bout, D. E. and White, M., Optimization by mean field annealing. In *Advances in Neural Information Processing Systems 1*, eds. Touretzky, D. S. Morgan Kaufmann, 1989, pp. 91–98.
28. Peterson, C. and Anderson, J. R., Neural networks and NP-complete optimization problems; a performance study on the graph bisection problem. *Complex Systems*, 1988, **2**(1), 59–89.
29. Peterson, C. and Söderberg, B., A new method for mapping optimization problems onto neural networks. *International Journal of Neural Systems*, 1989, **1**(1), 3–225.

30. Peterson, C. and Anderson, J. R., A mean field learning algorithm for neural networks. *Complex Systems*, 1987, **1**, 995–1019.
31. Bertoni, A. and Dorigo, M., Implicit parallelism in genetic algorithms. *Artificial Intelligence*, 1993, **61**(2), 307–314.
32. Grefenstette, J. J. and Baker, J. E., How genetic algorithms work: a critical look at implicit parallelism. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, Vol. 70–79. Morgan Kaufmann, VA, 1989.
33. Mühlenbein, H., Schleuter, M. G. and Krämer, O., Evolution algorithms in combinatorial optimization. *Parallel Computing*, 1988, **7**(1), 65–88.
34. Pettey, C. C., Leuze, M. R. and Grefenstette, J. J., A parallel genetic algorithm. In *Genetic Algorithms and Their Applications: Proceedings of the 2nd International Conference on Genetic Algorithms*. Lawrence Erlbaum, Cambridge, MA, 1987, pp. 155–161.
35. Tanese, R., Parallel genetic algorithm for a hypercube. In *Genetic Algorithms and Their Applications: Proceedings of the 2nd International Conference on Genetic Algorithms*. Lawrence Erlbaum, Cambridge, MA, 1987, pp. 177–183.
36. Collins, R. J. and Jefferson, D. J., Selection in massively parallel genetic algorithms. In *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, San Diego, CA, 1991, pp. 249–256.
37. Manderick, B. and Spiessens, P., Fine-grained parallel genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*. Morgan Kaufmann, VA, 1989, pp. 428–433.
38. Schleuter, M. G., Comparison of local mating strategies in massively parallel genetic algorithms. In *Proceedings of Parallel Problem Solving by Nature 2*, eds. Männer, R. & Manderick, B. Elsevier, Brussels, Belgium, 1992, pp. 65–74.
39. Grefenstette, J. J., Parallel adaptive algorithms for function optimization. Technical Report NCS-81-19, Vanderbilt University, Computer Science Department, Nashville, U.S.A., 1981.
40. Leite, J. P. B. and Topping, B. H. V., Improved genetic operators for structural engineering optimization, CIVIL-COMP95. In *Proceedings of the 6th International Conference on Civil and Structural Engineering Computing: Developments in Computational Techniques for Structural Engineering*. In *Developments in Neural Networks and Evolutionary Computing for Civil and Structural Engineering*. Civil-Comp Press, Cambridge, U.K., 1995, pp. 151–165.
41. Fogarty, T. C. and Huang, R., Implementing the genetic algorithms on transputer based parallel processing systems. In *Proceedings of Parallel Problem Solving by Nature*, eds. Schwefel, H-P. & Männer, R. Springer, Dortmund, FRG, 1990, pp. 145–149.
42. Neuhaus, P., Solving the mapping-problem experiences with a genetic algorithm. In *Proceedings of Parallel Problem Solving by Nature*, eds. Schwefel, H-P. & Männer, R. Springer, Dortmund, FRG, 1990, pp. 170–175.
43. Wright, S., Stochastic process in evolution. In *Stochastic Models in Medicine and Biology*, eds. Gurland, J. University of Wisconsin Press, U.S.A., 1964, pp. 199–241.
44. Shonkwiler, R., Mendivil, F. and Deliu, A., Genetic algorithms for 1-D fractal inverse problem. In *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, San Diego, CA, 1991, pp. 495–501.
45. Shonkwiler, R., Parallel genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms*. Morgan Kaufmann, IL, 1993, pp. 495–501.
46. Topping, B. H. V. and Leite, J. P. B., *Parallel Genetic Models for Structural Optimization*, accepted for publication in Engineering Optimization, 1997.
47. Mansour, N. and Fox, G. C., A hybrid genetic algorithm for task allocation in Multz computers. In *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, San Diego, CA, 1991, pp. 466–473.
48. Mühlenbein, H., Schomisch, M. and Born, J., The parallel genetic algorithm as function optimizer. In *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, VA, 1991, pp. 271–278.
49. Talbi, E.-G. and Brassière, P., A parallel genetic algorithm for the graph partitioning problem. In *ACM International Conference on Supercomputing*, Cologne, Germany, 1991.
50. Jog, P., Suh, J. Y. and Gucht, D. V., Parallel genetic algorithms applied to the travelling salesman problem. *SIAM Journal of Optimization*, 1991, **1**(4), 515–529.
51. Lin, S.-C., Punch, W. F. and Goodman, E. D., Coarse-grain parallel genetic algorithms: categorization and new approach. *Parallel and Distributed Processing*, Dallas, TX, 1994 (accepted for publication).
52. Punch, W. F., Averill, R. C., Goodman, E. D., Lin, S.-C., Ding, Y. and Yip, Y. C., Optimal design of laminated composite structures using coarse-grain parallel genetic algorithms. *Computer Systems in Engineering*, 1994, **5**(4-6), 415–423.
53. Adeli, H. and Cheng, N-T., Concurrent genetic algorithms for optimization of large structures. *ASCE Journal of Aerospace Engineering*, 1994, **7**(3), 276–296.
54. Adeli, H. and Kummar, S., Distributed genetic algorithm for structural optimization. *ASCE Journal of Aerospace Engineering*, 1995, **8**(3), 156–163.
55. Adeli, H. and Kummar, S., Concurrent structural optimization on massively parallel supercomputer. *ASCE Journal of Structural Engineering*, 1995, **121**(11), 1588–1597.
56. Khan, A. I. and Topping, B. H. V., Parallel adaptive mesh generation. In *Computing Systems in Engineering*, Vol. 2, no. 1. Pergamon Press, U.K., 1991, pp. 75–102.
57. Topping, B. H. V. and Cheng, B., Parallel adaptive quadrilateral mesh generation. In *Advances in Computational Structures Technology*. Civil-Comp Press, Edinburgh, U.K., 1996.
58. Khan, A. I. and Topping, B. H. V., Sub-domain generation for parallel finite element analysis. In *Computer Systems in Engineering*, Vol. 4, no. 4–6. Pergamon Press, U.K., 1993, pp. 473–488.
59. Rumelhart, D. E., Hinton, G. E. and Williams, R. J., Learning internal representation by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1: Foundations, eds. Rumelhart, D. E. & McClelland, J. L. MIT Press, U.S.A., 1986.
60. Khan, A. I., Topping, B. H.V and Bahreininejad, A., Parallel training of neural networks for finite element mesh generation. In *Neural Networks and Combinatorial Optimization in Civil and Structural Engineering*. Civil-Comp Press, Edinburgh, U.K., 1993, pp. 81–94.
61. Topping, B. H. V. and Bahreininejad, A., *Neural Computing for Sturctural Mechanics*. Saxe-Coburg, Edinburgh, U.K., 1996.
62. Topping, B. H. V. and Sziveri, J., Parallel sub-domain generation method. In *Developments in Computational Techniques for Structural Engineering*. Civil-Comp Press, Edinburgh, U.K., 1995, pp. 449–457.
63. Sziveri, J., Cheng, B., Bahreininejad, A., Cai, J., Thierauf, G. and Topping, B. H. V., Parallel quadrilateral subdomain generation. In *Advances in Computational Structures Technology*. Civil-Comp Press, Edinburgh, U.K., 1996.
64. Baffes, P. T., *Nets User's Guide*, Version 2.01, NASA, Lyndon B. Johnson Space Center, U.S.A., 1989.

65. Topping, B. H. V. and Khan, A. I., Parallel computation schemes for dynamic relaxation. In *Engineering Computations*, Vol. 11. Pineridge Press, Swansea, U.K., 1994, pp. 513–548.

66. Khan, A. I. and Topping, B. H. V., A transputer routing algorithm for non-linear or dynamic finite element analysis. In *Engineering Computations*, Vol. 11. Pineridge Press, Swansea, U.K., 1994, pp. 549–564.