ELSEVIER

Journal of Systems Architecture 42 (1996) 19–36

JOURNAL OF SYSTEMS ARCHITECTURE

Accelerating genetic algorithm computation in tree shaped parallel computer

Timo Hämäläinen [a,*], Harri Klapuri [a], Jukka Saarinen [b], Pekka Ojala [c], Kimmo Kaski [a]

[a] *Electronics Laboratory, Tampere University of Technology, P.O. BOX 692, FIN-33101 Tampere, Finland*
[b] *Signal Processing Laboratory, Tampere University of Technology, P.O. BOX 553, FIN-33101 Tampere, Finland*
[c] *Sierra Semiconductor, 2075 N. Capitol Avenue, M / S 19, San Jose, CA 95132, USA*

Received 2 June 1995; revised 1 February 1996; accepted 7 March 1996

Abstract

Realizations of genetic algorithms (GAs) in a tree shape parallel computer architecture are presented using different levels of parallelism. In addition, basic models for parallel GAs are considered. The tree shape parallel computer system, GAPA (Genetic Algorithm Parallel Accelerator) with special hardware for GA computation, is described in detail. Also mappings for centralized and distributed GA models are given and their performance has been measured for different population sizes.

Keywords: Parallel genetic algorithms; Parallel implementation; Parallel computing; Tree shape architecture

1. Introduction

Genetic algorithms are one of the most interesting novel methods for reducing the search effort and time in optimization tasks [1,2]. The basically simple concept of artificial evolution is very flexible and can be applied to a variety of problems, including those that are hard to model mathematically. However, applications may represent very different problem domains and therefore the style of GA may differ significantly from one application to another [3,4]. In general, GAs are strongly application dependent, which results in an abundance of their variations and extensions. In addition to the requirements of the problem at hand, it is important for their usage how to enhance the quality of the search and how to reduce the search time. Some improvements can be obtained by adjusting the algorithm and software, but for example further reduction in the search time requires also improvements in the computing hardware.

* Corresponding author. Email: timoh@ele.tut.fi.

PII S1383-7621/0165-6074(96)00009-4

First, experiments and realizations of GAs were performed in sequential computers, in which genetic operations were repeated numerous times in sequential loops. These systems are still widely used because of established programming tools, fairly good cost/performance ratio and ease of use. Unfortunately, the search time with them may still be too long for large problems or in real time applications. In addition, the parallel nature of GAs can not be fully utilized in sequential computers. For these reasons parallel computation is the most promising way for efficient GA implementations and consequently several parallel genetic algorithms and parallel realizations have been introduced [1,7–9]. Since GAs are very problem dependent, general purpose parallel computers, e.g., Transputers, Hypercube, MasPar and Connection Machine [11–14] have been preferred for flexible mapping of the problem. Especially transputers have been favoured, because the overall system can be built easily with inexpensive, modular chips. Other general purpose systems typically offer a number of processors and some programmable interconnection network. However, dedicated parallel hardware may offer even better performance than general purpose systems, since faster operation can be realized by implementing functions or even whole phases of an algorithm in hardware. On the other hand, dedicated hardware offers better cost-effectiveness, since all features of a large general purpose system are not needed for a particular task.

In this paper, we introduce our GAPA (Genetic Algorithm Parallel Accelerator) system, in which features of flexible operation of general purpose systems and fast processing capability of dedicated systems have been combined. Flexible operation is due to expandable tree shape architecture and programmable processing units, which ensure that several types of GAs can be mapped to the system. Good cost/performance ratio has been achieved by using commercial and common Digital Signal Processors (DSPs) and Field Programmable Gate Array (FPGA) chips.

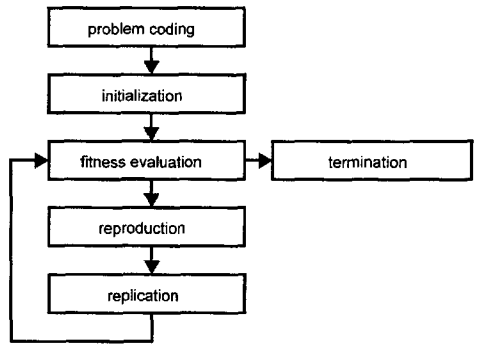


Fig. 1. Operations in a typical GA.

Before we go to details of our GAPA system, we first discuss a typical sequential genetic algorithm and the most time consuming phases in it. After that we introduce different levels of parallelism in GAs and how to utilize them in parallel computation. We also present various models for parallel GA and their implementation styles in parallel computers. The implementation in a tree architecture is given in more detail, whereafter we consider the structure and function of the GAPA system, including special hardware. Finally, we show how to map different types of GAs to the GAPA and conclude with discussion about its performance and future work.

2. Sequential genetic algorithm

The flow of computation in a typical sequential GA is illustrated in Fig. 1. In the first phase the problem is encoded to an individual. The actual computation starts in the next phase, in which a number of randomly generated individuals are created to form an initial population. After that, the fitness of each individual is evaluated and if a sufficient solution is found, the GA execution is terminated. Otherwise, some method is used to select parents among individuals for the reproduction phase. In this phase a new generation of the population is

created or the old population is updated, and previous phases are repeated.

The loop of fitness evaluation, reproduction and replication may be performed a number of times before a satisfactory solution has been found. The more time one phase consumes, the larger will be the total search time. On the other hand, the larger the population and the longer the individual strings, the more time each phase takes. Although the time needed for each phase is dependent on the problem as well as computing platform, some general estimations can be given. The initialization phase is needed only once at the beginning of the execution and thus it has minor contribution to the total search time. In contrast, the fitness evaluation may be very complicated, so that it consumes most of the total execution time. The reproduction phase consists of selection of mates and string manipulations. The selection scheme may be complicated and requires many random numbers, for which reason this phase may also be time-consuming. String manipulations take least time, especially when strings are binary valued.

In order to reduce the total search time one has to try to improve the search quality and speed up the execution of single phases. The search quality can be adjusted by parameters like mutation rate and crossover probability [19]. Since this is beyond the scope of this paper, we concentrate on the speed-up of computation. In the next section we show how this can be achieved by means of parallel processing.

3. Parallel genetic algorithms

The outline of sequential GAs discussed in the previous section illustrates the core sequence of high level operations, which have to be performed one after another. However, we can utilize parallel processing in each of these phases by parallelizing operations at several levels. We can also parallelize the high level structure itself by executing several GAs in parallel. Therefore, a hierarchy of parallelism levels can be composed: *logical operations level, phase level, population level and process level parallelism.*

At the lowest level the basic arithmetic and logical operations can be parallelized. For example, all bits of a long string are processed in parallel. At the phase level parallelism, the whole phase of an algorithm is parallelized, so that there exist no longer any sequential computation within that phase. The fitness evaluation is a good example of such a phase. Because the fitness of an individual is not dependent on the fitness of other individuals, all individuals can be evaluated in a truly parallel fashion. These two lowest levels of parallelism can be utilized in sequential GAs, without changing the high level structure of the algorithm.

At population level parallelism, the GA may be parallelized by dividing the original population into many subpopulations. These are guided to evolve and periodically some of the best individuals are exchanged between subpopulations. Practically this means, that there are many copies of the GA running in parallel. At the process level there may be again many GAs executed in parallel, but now they can be different and independent of each other. The population may be the same for all processes, which means that the solution is searched among many separate processes for the same problem.

The two highest levels of parallelism are the basis for parallel genetic algorithms. Usually parallel GAs have features from several levels of parallelism, for example population level parallelism may be combined with phase level parallelism. The realized parallelism depends naturally on the computing platform and its features. It should be noted, that parallel GAs have also been simulated in sequential computers, but as mentioned earlier, this may lead to an inefficient implementation. Independent of the implementation style, three main categories for parallel GAs can be found, i.e., *centralized, distributed and network models* [5,6,12].

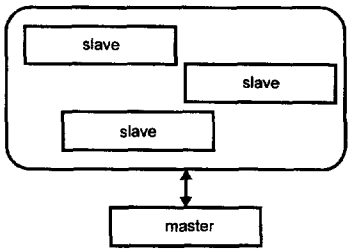


Fig. 2. Centralized or master-slave implementation.

The centralized style of parallel GA is illustrated in Fig. 2. The idea is to perform some basic calculations or phases of the sequential GA in many slave processors, whereas the master processor maintains the actual population. For that reason it is also called the *master-slave or farming model* [1,15]. In this approach, the master sends one or more individuals to each slave for fitness evaluation and collects results from them. After that the master can perform other phases of the algorithm and again send new individuals for fitness evaluation.

Distributed GA is a direct application of the population level parallelism, as depicted in Fig. 3. This is sometimes called an *island model* [7], since the population is divided into many subpopulations, each of them living in its own "island". In this case, a host is needed to control the exchange of individuals between subpopulations as well as termination of execution when the solution has been found.

A network model can be derived from the distributed implementation by increasing the number of subpopulations such that there is only one individual

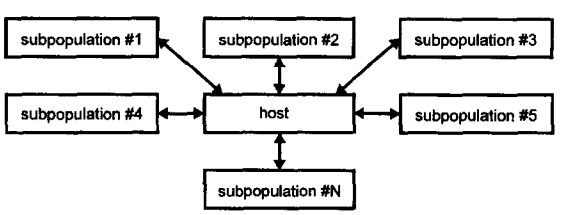


Fig. 3. Distributed implementation.

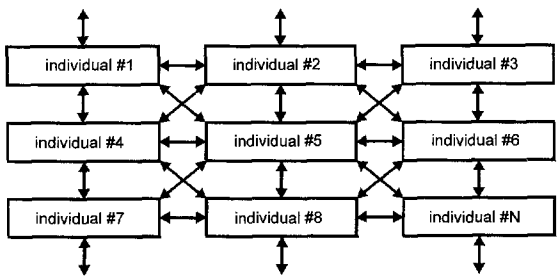


Fig. 4. Network model implementation.

per island. In addition, the individuals are physically connected together as shown in Fig. 4. Possible interconnection network topologies are for example two dimensional grid, hypercube, ring and ladder [6,10,11]. In this model each individual is living on its own site and occasionally interacts with some of its nearest neighbors. In other words, with some probability an individual mates randomly with one of its neighbors and may be replaced by the resulting offspring. The selection is performed locally without knowledge of globally best individuals. This model is also called *diffusion model*, because "good" individuals are effectively diffused around the topology of individuals during execution.

All these parallel GA models can be implemented in parallel computer systems. The centralized model is well suited for Single Instruction stream Multiple Data streams (SIMD) computer architecture, because it offers centralized and simple control over the system. The network model may be mapped to a Multiple Instruction streams Multiple Data streams (MIMD) computer [18]. In this case, each individual may be placed on its own processor and processors may be run independently. The distributed model may be implemented in either one of these architectures, depending on the resources available in the system. There is also many possible topologies for arrangement of processors in these architectures, resulting many choices for GA computation. Although the optimal mapping depends on the target computer

system, some general styles of mapping can be given.

The centralized GA is well suited to an array of processors connected to a master processor via bus. This is reasonable, because the master can not send or collect individuals simultaneously from slave processors. In the case of distributed GA, each subpopulation is best mapped to a single processor and one of the processors may act as a host. The physical connection between processors does not need to be the same as logical connection between subpopulations. The more important thing is, that an individual can be routed to the desired subpopulation by some means. In the network model GA, each processor may be assigned one individual. Usually this means that the topology of processors also dictates the topology of individuals, because there is intensive communication between processors so that communication distances should be minimized.

The parallel implementation of GAs raises also problems, which are not present with sequential computers. The granularity of the parallelized algorithm should be suitable for efficient utilization of the processors and communication network. Fine granularity means that very simple calculations are performed in each processor and there might be large amount of communication between processors. In contrast, coarse granularity will require complicated operations or much temporary memory for processors. Another problem is how to balance the workload between processors. If the granularity of the algorithm is fine, the workload might be well balanced, but still the performance is reduced because of communication overheads. On the other hand, when the granularity is too coarse, some processors may have much idle time.

The control of computation should be considered carefully. Synchronous operation of processors and tight global control ensures good management overall, but in turn may cause overheads for some processors. For this reason also asynchronous implementations of parallel GA models have been introduced [1]. An example would be an *asynchronous master-slave* implementation, where slaves are computing fitnesses independent of each other and even while the host is performing other phases of the algorithm. In this case there might be individuals from many generations simultaneously and there no longer exist clear boundaries between old and new populations. This is sometimes called *continuous generation* implementation.

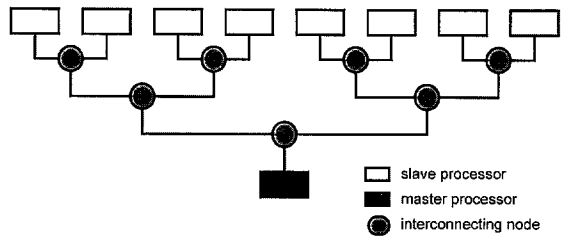


Fig. 5. Tree shape multiprocessor architecture.

In summary, there are many possibilities for parallel GA implementations and their mappings to several parallel hardware platforms. In the next section we introduce how genetic operations can be mapped to a tree shape computer architecture.

4. Tree shape architecture in GA computation

The tree shape parallel computer architecture is illustrated in Fig. 5 [18]. The top of the tree is formed by a horizontal line of processors, the trunk is composed of interconnecting nodes and the root is either a processor or an interface to another system. In general, this architecture can be referred to as master-slave configuration, where the root acts as a master and the top processors as slaves. This observation is confirmed by the way of communication, since there is no direct connection between processors and all the communication should be carried via the root. In the basic form the trunk of the tree is only a routing resource, where the interconnecting

nodes behave as switches to establish a connection between processors and the root. However, the role of the trunk (or communication network) may be extended, if some advanced operations could be included in interconnecting nodes. By this way, the communication network as a whole becomes an active processing element. As processors on top of the tree are dedicated to local operations, the communication network may be used for global operations such as broadcasting, summation, string manipulation or sorting. These features can be utilized at all levels of parallelism, depending on the granularity of the GA.

In order to illustrate the possible roles of elements in the tree, let us consider a mapping of some GA operations to this architecture. The fitness evaluation is most conveniently performed in processors, since the fitness function is always dependent on the given problem and processors can be programmed flexibly. Then there are many possibilities to map reproduction phase to the architecture. Especially the selection of mate pairs for crossover and mutation can be done in several ways. In some selection methods the population should be sorted, which may be performed in the communication tree. Processors send individuals for comparison and the master or host receives a list of sorted individuals, as illustrated in Fig. 6. In the familiar roulette-wheel selection method

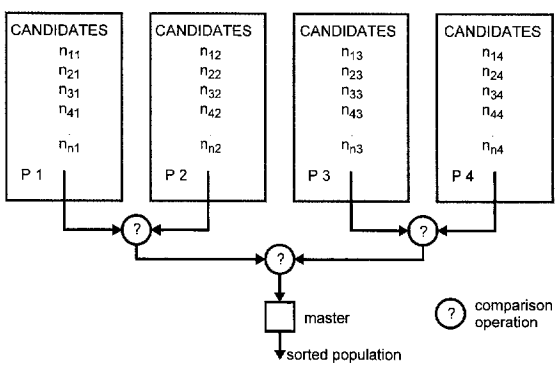


Fig. 6. Sorting operation in the tree.

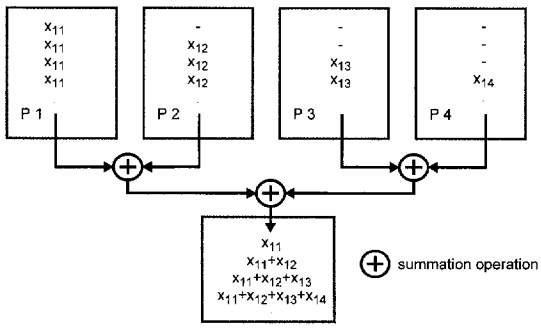


Fig. 7. Cumulative summation in the tree.

a cumulative sum of fitness values of individuals is required. This can be again computed in the communication network, since each interconnecting node in the tree can be configured as an adder, as depicted in Fig. 7. In both of these cases the master selects the mate pairs. Selection can also be performed in slave processors, if the selection is based on local knowledge of fitness values, as is the case with network model GAs.

The crossover and mutation phases consist of quite regular string manipulation tasks and can be performed very efficiently in hardware. Now the communication network can be utilized by performing crossover and mutation in interconnecting nodes, as illustrated in Fig. 8. The execution starts by first sending parents from processors down to the communication network. In some interconnecting node the parents meet for crossover and mutation. The resulting offspring or offsprings are then passed fur-

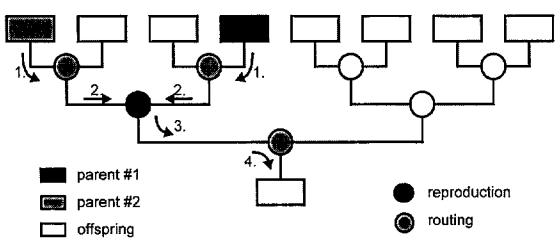


Fig. 8. Reproduction in the tree.

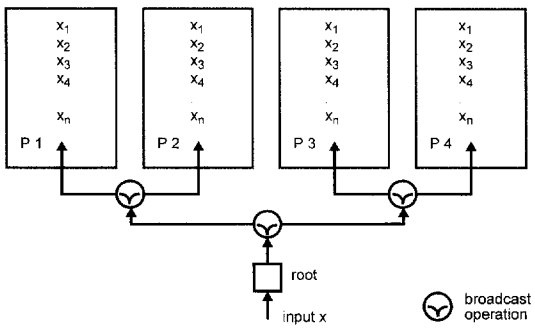


Fig. 9. Broadcast operation.

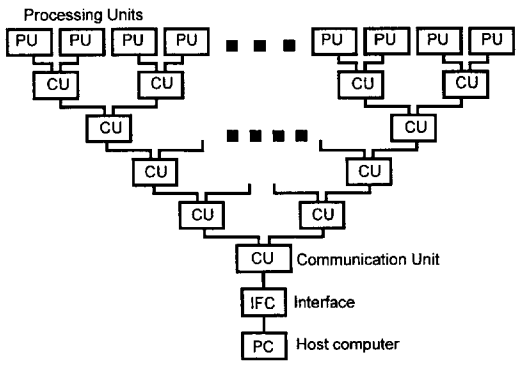


Fig. 10. Architecture of GAPA computer system.

ther to the master. Reproduction in the tree offers also an interesting variation to this basic scheme of two mating parents. A *multiparent mating* results, if all processors send individuals to the tree simultaneously and the reproduction is performed in all nodes. Naturally the crossover and mutation can be performed also in the master. Slave processors should be used, if the selection is based on local set of individuals. In addition to these mappings, the broadcast feature of the communication network can be utilized in all phases, where the same data should be delivered to all processors. The broadcast mode of the tree is depicted in Fig. 9.

The essential question concerning these mappings is, how efficiently global operations in the tree can

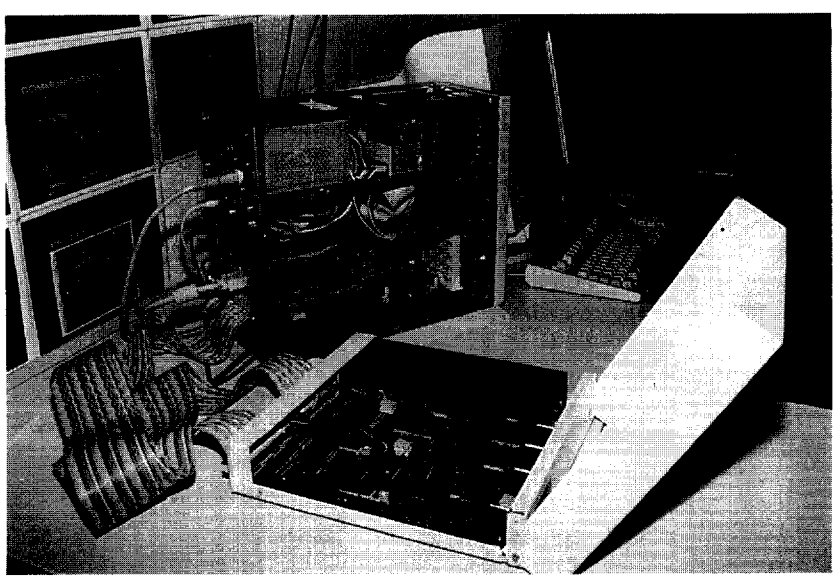


Fig. 11. Photograph of GAPA prototype system.

be performed. In the next section we describe the GAPA system in detail to see how the performance of execution can be enhanced by special hardware.

5. GAPA computer system

The GAPA system is based on the TUTNC general purpose neurocomputer, which was designed at Tampere University of Technology in 1993 [20,21]. The GAPA system utilizes the same tree shape architecture at high level, but the structure and function of basic elements are different due to reconfigurability of the hardware. The overall architecture is shown in Fig. 10. The processors are named processing units (PUs), the interconnecting nodes in the trunk are called communication units (CUs) and the root is an interface (IFC) to a host computer. All PUs or CUs are identical to ensure modularity and simple implementation. The system can be easily expanded by adding PUs and branches to the communication network. Based on these architectural considerations, a small scaled prototype of four PUs and three CUs has been realized as shown in Fig. 11. In the prototype the main components of the system are a processor board and the host computer, which is an IBM PC compatible personal computer. The actual tree shape architecture of PUs and CUs is built in the processor board. In the following we describe in more detail the structure and function of the system. We start by describing the functions of the system. After that we explain the structure of PUs and CUs and especially the dedicated hardware for GA computation.

The basic modes of communication in the tree are broadcast, write and read. The host can deliver data to all PUs, write data to a particular PU or read data from one PU at a time. The data processing is included in the extended modes of communication, that are summation, comparison and reproduction, and they are linked to the general way of processing in the GAPA, as illustrated in Fig. 12. The processing can be modelled as a folded pipeline, which is composed of the processes performed by the host, the communication network and the PUs.

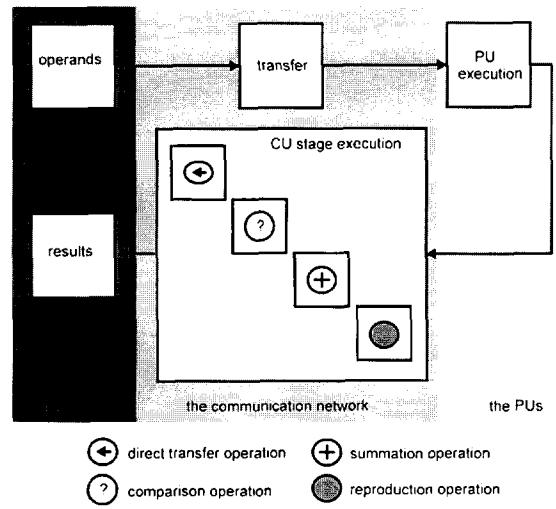


Fig. 12. Operational view of GAPA system.

The processing units are actually computers of their own, each equipped with a local memory and a general purpose digital signal processor (DSP), as shown in Fig. 13. The DSP is Texas Instruments' TMS320C25 [22], which was selected for its low cost, good availability and sufficient performance. The program code for the DSP is loaded in the local

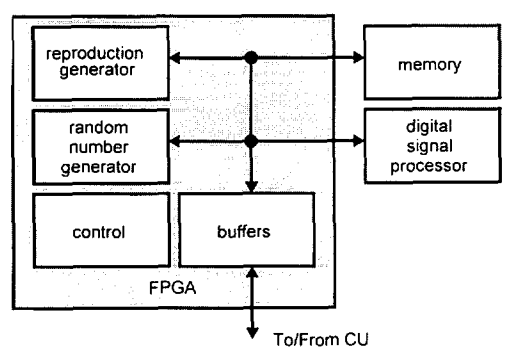


Fig. 13. Block diagram of processing unit.

memory, which is also used for storage of run-time variables. The memory and DSP are connected to the communication network via interfacing circuitry implemented to a single XILINX XC4005 FPGA chip [23]. In addition, the FPGA chip is equipped with a parallel random number generator and special reproduction generator, which can be accessed by the DSP. In other words, the FPGA chip acts as a coprocessor for the DSP. These special pieces of hardware are aimed to accelerate string manipulations in the reproduction phase of a GA, if this phase is executed in the PU. Since random numbers are required frequently during GA execution, it is useful to accelerate their generation by hardware. The random number generator can operate continuously so that the DSP can read a new random number at any time it is needed. The task of the reproduction generator is to get parents from the DSP, perform crossover and mutation and send offsprings back to the DSP. It should be noted, that this generator is suitable only for binary strings.

The structure of one CUs is depicted in Fig. 14. CUs are routing switch elements with reduced set of arithmetic and logical functions. In the basic configuration one CU can perform comparison and summation operations in its arithmetic-logical unit (ALU).

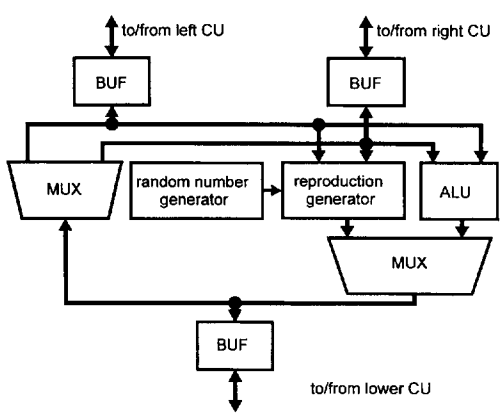


Fig. 14. Block diagram of communication unit.

The random number generator and reproduction generator can also be included to a CU, if the reproduction is performed in the tree. Data can be routed from the bottom arm to either one of the top arms or shared between the left and right arms. In the opposite direction, data can be routed from either of the top arms to the bottom arm. In the case of summation, operands are fetched from the top arms and the result is passed down to the next CU or host. In the sorting operation candidates from the top arms are compared and the winner is let to travel down in the tree. The reproduction can be performed such, that parents from the top arms are mated in the reproduction generator and resulting children are passed forward. CUs are implemented with XC4005 FPGA chips, which allow easy regonfiguration for desired functions. Data is buffered and clocked in each CU, therefore CUs can also be seen as pipeline stages. All routing or processing functions can be performed in a single system clock cycle.

As was mentioned, both PUs and CUs can be equipped with special hardware to accelerate GA computation. Reproduction generator and random number generator implement calculation level parallelism, which means that they are executed at very low hardware level. At that level there is no longer abstract symbols or strings of characters, but rather a composition of Boolean logic operations for strings of bits. In the following we clarify the structure and function of these special generators at that hardware level. Since the reproduction is performed slightly differently in PU and that in CU, their structure also differ.

The reproduction generator for the PU is shown in Fig. 15. For parallel string manipulation, there is a set of registers, which can be accessed by the DSP and multiplexers. Parents to be mated are written to the parent registers and the offspring can be read from its own register. In addition there is two mask registers, which control the crossover points and mutation of the offspring. Each bit position of the offspring register is connected to a 2-input multi-

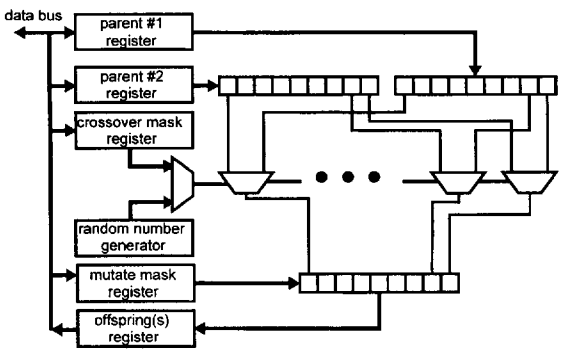


Fig. 15. Reproduction generator for processing unit.

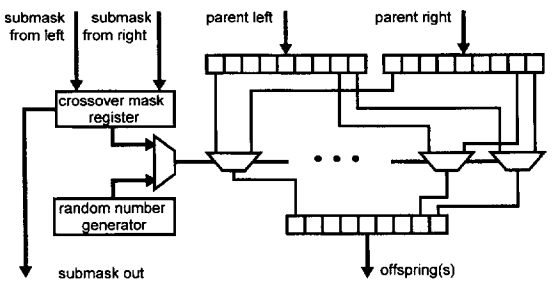


Fig. 16. Reproduction generator for communication unit.

plexer, such that a particular bit can be fetched from either one of the corresponding bit positions in parent strings. The multiplexer is controlled by the corresponding bit in the crossover mask register. Another possibility is to control multiplexers by a random number, which can be connected to multiplexers instead of the mask register. By this way the crossover can be performed in truly parallel fashion. The crossover type and number of crossover points can be determined very flexibly using mask register, or randomly using random number generator. The mutation is performed by inverting random bits in the offspring string. This can be done during operation by assigning desired points of mutation to the mutation mask register. From the DSP point of view, the whole crossover and mutation phase is only composed of writing and reading parent and offspring registers and possible computation of masks. The registers may be in some cases designed to be as long as individual strings, meaning that there is no need to divide individuals into pieces for string manipulation.

The structure of the reproduction generator for CUs is a little different from that designed for PUs, as can be seen in Fig. 16. Now the parent strings are fetched from the top arms along with possible submasks. The idea is to combine the final crossover mask from the submasks, which have been originally computed and sent by different PUs. Another choice is to use random number generator to control the crossover, if the mask scheme is not used. If there is only one parent present in either top arm, no reproduction is performed and the parent is only passed down. In the case of reproduction only the offspring is passed down. The mutation operation can be performed in the host computer or autonomously in the CU, which means that no mutation mask is sent by any PU. Instead, CUs are commanded to produce random numbers and invert occasionally bits of the offspring string.

The random number generator can be used as stand alone or in conjunction with the reproduction generator, as mentioned earlier. In our system random numbers are generated in Tausworthe pseudorandom number generator, based on feedback shift register, as shown in Fig. 17 [16,17]. It produces uniformly distributed random numbers that have minimum pair correlations in a long sequence. If the length of the shift register is P, the recurring linear

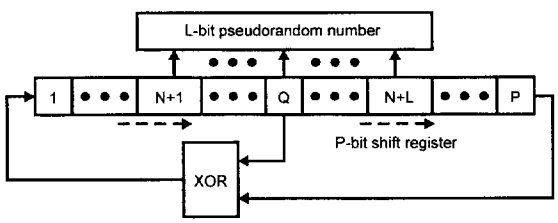


Fig. 17. Feedback shift register method for pseudorandom number generation.

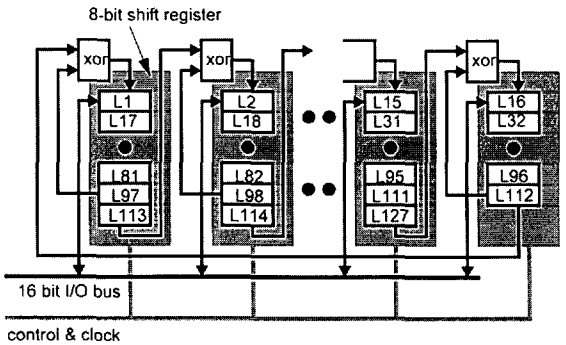


Fig. 18. Parallel random number generator.

sequence of numbers is at maximum 2^{P-1}, provided that the feedback point Q and shift register length P are selected properly. The operation of the random number generator is the following. As a first step the shift register is initiated by assigning randomly chosen seed number. In the next clock cycle bits are shifted to the right and to the first bit position, a modulo-2 sum (i.e. XOR) of the two feedback bit positions Q and P is inserted. The contents of the last bit at position is lost after each shift operation. By this way there are L shifts and XOR operations required for each new L bit random number. Sometimes even this takes too much time and cannot be tolerated. Faster operation can be achieved by making the modulo additions concurrently, which yields a new random number in each clock cycle. This idea is implemented in our hardware random number generator, illustrated in Fig. 18. In this implementation, there are many short shift registers and a network of XOR elements such that a single shift operation is equivalent to L shifts in the original generator. The new random number can be read from the first bit positions of the shift registers via an interface bus (see Fig. 18). The bus is also used to feed seed numbers to the generator during initialization phase. In Fig. 18, the random number is selected to be 16 bits wide and values for feedback positions are $Q = 15$ and $P = 127$.

An important issue considering this generator is the initialization phase. Each random number generator in PUs and CUs should produce uncorrelated random number sequences, which means that PU or CU may use only distinct pieces of the available random number sequence. For this reason the first generator can be initiated by a random seed number, but the others have to be initiated based on this first seed number. If there are e.g. four generators, the available random number sequence is partitioned into four subsequencies. Each generator is then initiated to start with one of the four starting points within the original sequence, and several methods have been suggested to solve this problem to determine seed numbers for each generator, see a review of Ref. [17].

6. Mapping of genetic algorithms to GAPA

The architecture and special hardware features of the GAPA offers several implementation possibilities. In the following, two different example mappings based on centralized and distributed GA models are presented. The flow of computation for the centralized mapping is illustrated in Fig. 19, and considered first. For both mappings, it is supposed that the total number of individuals is N and available PU count is P. Individuals are implemented in binary strings of L bits. In centralized mapping the whole population is delivered to PUs such, that there are N/P individuals in each PU. The computation starts by creating initial individuals to each PU by generating random numbers. Since each random number fetched from the random number generator is 16 bits wide, $(N/P)\lceil L/16\rceil$ numbers are needed in each PU. After fitness evaluation, the best individual is searched and compared to the termination criteria. If the criteria is not met, the execution continues to the reproduction phase. Mates S_i^P and S_j^P are selected such that

$$\begin{cases} S_i^p, & p = [1,P],\ i = best^p, \\ S_j^p, & p = [1,P],\ j = rand(1,N/P). \end{cases} \quad (1)$$

Mating is repeated N times, because only one offspring is produced during the mating operation. The reproduction is performed in the communication network, where the crossover is controlled by the random number generator (see Fig. 16). The host collects new individuals and writes them to PUs, which finishes the evaluation of the generation.

The flow of computation for the distributed mapping is illustrated in Fig. 20. In this mapping, populations are separate and evolve independently in each

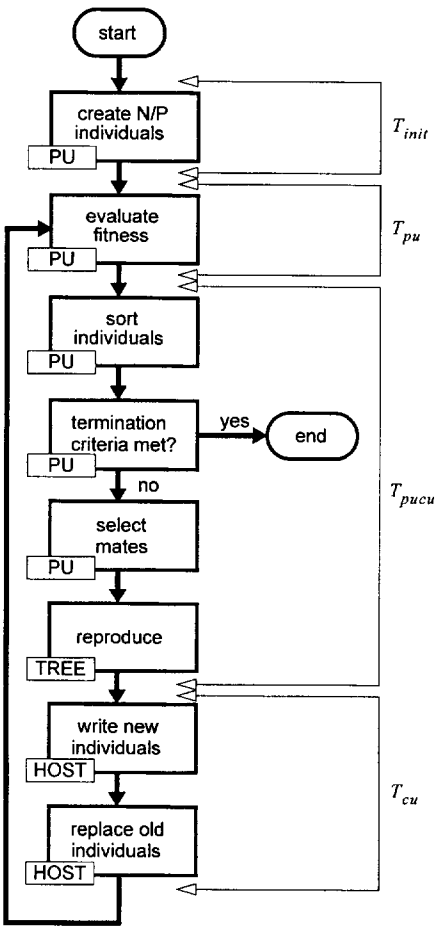


Fig. 19. Execution flow of centralized implementation.

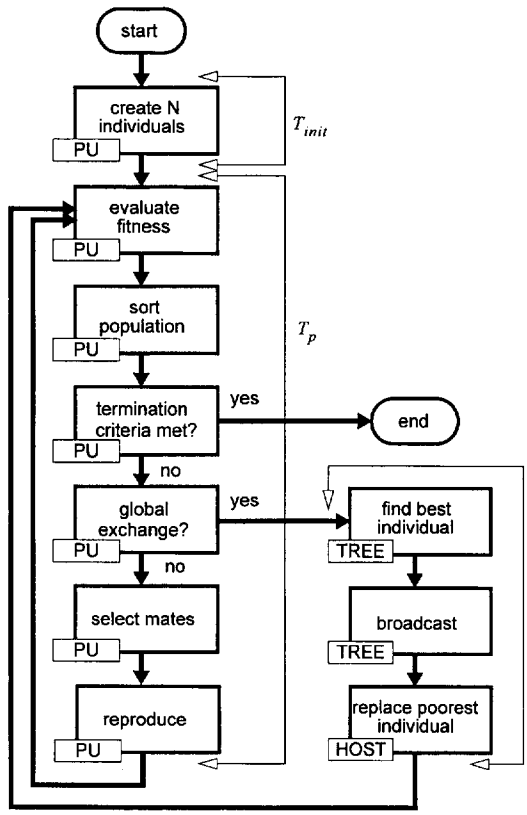


Fig. 20. Execution flow of distributed implementation.

PU. Initialization and fitness evaluation phases are similar to the centralized mapping. The selection of mates, however, is slightly different and proceeds as follows. First, the population is sorted from best to poorest according to the fitness. The first mate S_i is selected from the list, beginning from the best individual. Only four best individuals are used in each mating operation. The second mate S_j is selected at random, i.e.

$$j = rand(0,N). \tag{2}$$

By this way the selection corresponds to the centralized mapping. The reproduction is performed in the reproduction generator, and the crossover is

controlled by the random number generator (see Fig. 15). Occasionally the host commands PUs to exchange individuals. Each PU sends its best individual for global comparison, and the globally best individual is broadcast to all PUs. In each PU, the poorest individual is replaced by the broadcast one.

Both mappings utilize random number generators as well as reproduction generators in a similar manner, although the structure of algorithms differ. Main differences are found in population arrangement and parameters, which affect the convergence in a given problem. However, since the scope of this paper is in hardware implementation, the mappings are not compared in this sense. In the next section we discuss performance issues to show, how special hardware accelerate GA computation.

7. Performance

In GA computation, the performance can be determined in a number of ways depending on the point of view. At the highest level, wall clock time for convergence in a problem can be given. This time depends on parameters for algorithm as well as speed of computation. In order to see different contributions, we consider first the performance of system components in GAPA. After that we study realized execution times for the previously presented mappings.

The average performance of the TMS320C25 DSP is 10 MIPS in 16 bit, fixed point arithmetic at 40 MHz operation frequency. Depending on the addressing mode and memory configuration, the theoretical I/O transfer rate is at maximum 52 Mbits/s (3 DSP instruction cycles per word required). In GAPA, both data and program code are executed in external memory with one additional wait cycle, resulting a maximum rate of 32 Mbit/s. It should be noted, that the DSP is clocked at 40 MHz, while the system clock rate is 10 MHz for all FPGA chips in GAPA. The performance of the reproduction and random number generator can be given in terms of clock cycles needed to accomplish a given task. For example, the reproduction in the PU can be finished in three system clock cycles, which is the time for two write and one read access to the parent and offspring registers. Clocked at 10 MHz, 3.3 million offsprings can be produced in a second. The realized rate for offsprings is much lower, since the DSP can not access the registers so fast. From the parallel random number generator a new number can be fetched in each system clock cycle. Thus 10 million random numbers per second can be generated.

The performance of the CU can be given as data transfer rate or operations per second. Thus, 360 Mbit/s data transfer rate through each CU or 10 MIPS performance for summation, comparison or reproduction can be given. In addition, pipelining results 30 MIPS peak performance for a communication network of three CUs. If we consider the reproduction phase in the tree, 10 million offsprings per second can be produced. The realized rate for offsprings, however, depends on the interface between GAPA and host computer. In the prototype system, the interface is designed for PC ISA bus, which can transfer data 64 Mbit/s.

It is noted, however, that the above numbers give the theoretical maximum performance. The true performance is obtained by measuring the phase times for each single operation. Fig. 21 illustrates the phase times for generation of one random number word (abbreviated RNG), reproduction of two word-length individuals (RPD) and comparison of four words in the communication tree (COMP) using special hardware features. For comparison purposes, the same phases are also performed without utilizing hardware generators and comparison in the tree. For example, random numbers are generated in software in the PU instead of random number generator. In comparison operation, the host first reads words from PUs and performs the comparison. From Fig. 21, it can be seen, that significant speedup is gained in each phase by using special hardware.

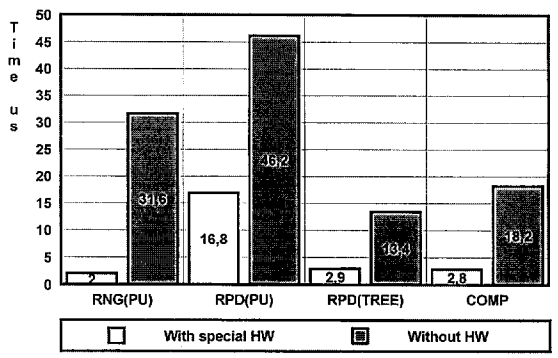


Fig. 21. Phase times. RNG(PU): the time for random number generation in PU, RPD(PU): the time for reproduction in PU, RPD(TREE): the time for reproduction in tree, COMP: comparison operation in the tree.

Next we consider the performance in computation of the centralized mapping by studying pairs of individual string length L and population size N; $(L,N) = (64,256)$, $(128,128)$, $(64,512)$, $(128,512)$ and $(256,256)$. This is to see what is the effect of population "shape" on execution times. For simplicity the fitness function has chosen to be $f = x^2$. Measured parameters are the time for PU computation phase (T_{pu}), the time for PUs and CUs computing concurrently (T_{pucu}) and the time for communication (T_{cu}), as illustrated in Fig. 19 during the flow of computation. T_{pu} refers to the phase, where only PUs are computing and CUs are idle, while T_{cu} refers to the communication phase. The reproduction phase requires both PUs and CUs to be active, for which reason the time for this phase is measured separately. Results are shown in Fig. 22 for the evolution of one population generation, where each column marked "A" shows the execution time for a mapping that utilizes special hardware generators. As can be seen, the PU computation takes the major part of the total execution time in long individual strings due to the fitness evaluation. In order to see the effect of special hardware to the execution times, centralized mapping has been designed also without the usage of hardware generators. Execution times for these mappings are labelled "B" in every second column of Fig. 22. As the hardware generators are used only in the reproduction phase, T_{pucu} is different in each pair of mappings. Since the computation of the fitness function takes most of the total execu-

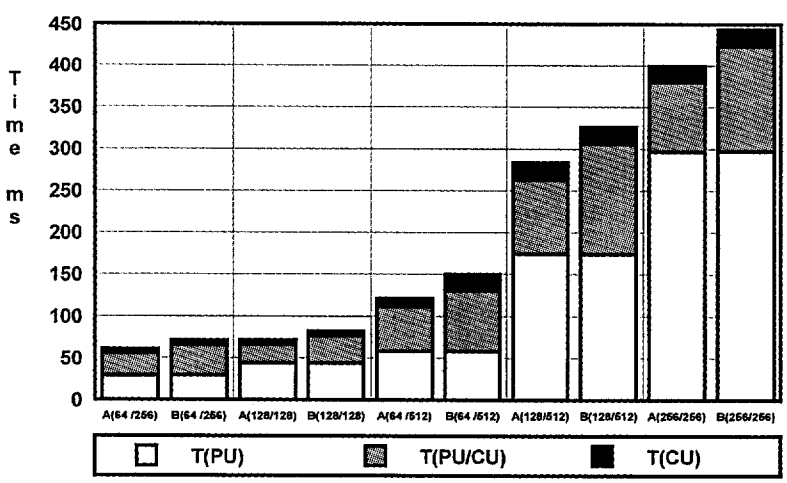


Fig. 22. Execution time for one generation in centralized GA. A) Special hardware utilized, B) Without hardware support. T(PU): execution time in PU, T(PU/CU): the execution time, when PUs and CUs are active simultaneously, T(CU): the time for communication.

tion time, the speedup in reproduction yields only a minor improvement as a whole. However, this is dependent on the population shape, such that the total speedup is the greater the more individuals there are for a given string length.

Until now we have seen how special hardware accelerates single phases of execution and due to this their contribution to total execution times. Comparisons between different computing platforms may be also fruitful. However, fair comparisons are very difficult to perform, because parameters may not be uniform from one implementation to another. For this reason we have mapped centralized and distributed GA for a PC and compared the execution times with those of GAPA. The PC used in comparisons is equipped with a 100 MHz Pentium processor, 32 Mbytes of RAM, and Microsoft Windows NT operating system. We have evaluated the performance of this PC to be about the same as SUN SparcStation 5. For the software platform we chose MATLAB 4.0, because it is widely used for prototyping purposes due to its easy programming and visualization features. Unfortunately, MATLAB interprets its m-file commands rather than compiles them, which increases execution times significantly. This can be overcome by a MATCOM program, which first compiles m-files into C + + code, and finally produces fast executable code using standard C + + compilers. Execution times have been measured for both original MATLAB simulations and compiled C + + versions of the corresponding m-code. A word of caution, however, should be given for comparisons with GAPA. In MATLAB, each bit position in an individual have to be implanted as a floating point number, such that a "0" or "1" in a position is actually a real number. In addition, the fitness value $f = x^2$ is computed by first decoding individual string to a real number and raising this number to the power of two. In GAPA, the fitness is computed by applying several times 16 bit, fixed point multiplications and summations for each individual string. Thus, the time for fitness evaluation is

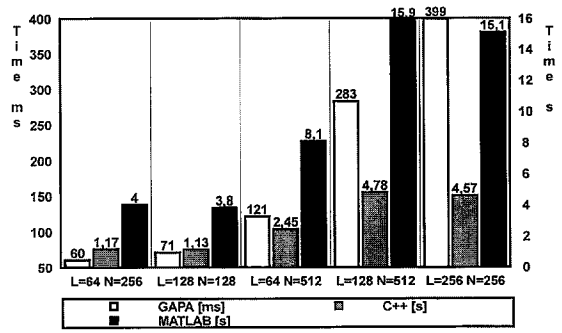


Fig. 23. Average execution times for one generation in centralized GA for GAPA, MATLAB and C + + mappings.

strongly dependent on the string length in GAPA, while in MATLAB this is not so obvious. This difference can be clearly seen in the total execution times, as depicted in Fig. 23. It should be noted, that the results for GAPA are given in microseconds, while results of MATLAB and C + + mappings are given in seconds. As can be seen, the shape of the population have only a minor effect on MATLAB and C + + implementations. For example, in cases $(L,N) = (128,512)$ and $(L,N) = (256,256)$ the corresponding GAPA execution times differ about 30%, while both MATLAB and C + + execution times differ about 5%. The population size in bits is the same for both cases, and conforms previous observation of differences in fitness function computation. In general, GAPA offers a minimum speedup of 10 fold compared to MATLAB and C + + mappings in these cases.

Next we consider the distributed GA implementation. In this case, the computation is performed mainly in PUs using special hardware in a similar manner as in centralized mapping. Thus we study only the total execution times, as follows

$$T_{total} = T_p + T_c, \tag{3}$$

where T_p is the time for PU computation and T_c the time for communication. Measure points for these parameters are given in Fig. 20. In practice, this

means that we study evolution of one generation with global exchange of the best individual. Execution times are given for $(L,N) = (64,1024)$, (128,512), (64,2048), (128,2048) and (256,1024), and the results are shown in Fig. 24 for GAPA, MATLAB and C + + mappings. Minimum speedup gained by GAPA is about 20 fold compared to C + + mapping.

8. Summary and conclusions

The architecture and implementation of a parallel hardware platform for GA computation have been introduced. The GAPA is a multiprocessor system with programmable processing units. The number of PUs can be easily increased, such that the desired degree of parallelism and performance can be reached. In addition, the system is modular, since all PUs and CUs are identical. Special hardware generators can be included in both PUs or CUs to accelerate GA computation at low level. Flexibility of operation is due to FPGAs and DSPs, which can be programmed and controlled individually. The cost of the implementation has been kept fairly low, since commercial, popular components have been used. The system concept has been tested with a small scale prototype of four PUs, and the architecture and the special hardware has been found to be well matched with requirements of different parallel GA models. Mappings of centralized and distributed models of parallel GAs have been presented. In addition, performances have been analyzed, and compared to corresponding implementations in a PC. Results show, that special hardware accelerate GA computation, and significant speedups can be obtained by GAPA compared to PC. However, the computation of the fitness function consumes most of the total execution time, and should be decreased in order to balance the phase times. Since the fitness function differs from one application to another, dedicated hardware is not reasonable to use. In GAPA, the current DSP can not handle data as fast as hardware generator or the communication network, thus introducing a bottleneck to the overall computation. On the other hand, DSP may not be an ideal for GA computation and more performance and flexibility may be obtained by some other processor

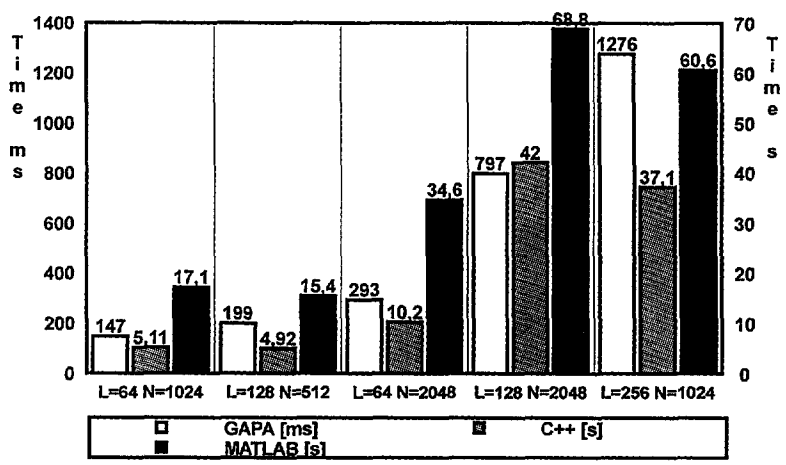


Fig. 24. Average execution time for one generation in distributed GA for GAPA and MATLAB.

type. For example, a Reduced Instruction Set Computer (RISC) style processor may be more suitable, and should be studied in the future. Also, the performance should be analyzed in more detail to see, how different selection schemes and fitness functions affect the figures of merit.

Acknowledgements

This research was supported by the Academy of Finland and Graduate School on Electronics, Telecommunications and Automations (GETA)-programme. We would like to thank Prof. Jarmo Alander for providing access to his comprehensive GA bibliography.

References

[1] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addision-Wesley, 1989.

[2] J. Holland. Genetic Algorithms. *Scientific American*, pp. 44–50, July 1992.

[3] Feng-Tse Lin, Cheng-Yan Kao and Ching-Chi Hsu. Applying the Genetic Approach to Simulated Annealing in Solving Some NP-Hard Problems. *IEEE Transactions on Systems, Man and Cybernetics* 23(6):1752–1767, November/December 1993.

[4] J. Cohoon, S. Hedge, W. Martin and D. Richards. Distributed Genetic Algorithms for the Floorplan Design Problem. *IEEE Transactions on Computer Aided Design* 10(4):483–491, April 1991.

[5] R. Tanese. Distributed Genetic Algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms, ICGA -89*, pp. 434–439, Georg Mason University, Morgan Kaufmann Publishers, 4–7 June 1989.

[6] B. Manderick and P. Spiessens. Fine-Grained Parallel Genetic Algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms, ICGA-89*, pp. 428–433, Georg Mason University, Morgan Kaufmann Publishers, 4–7 June 1989.

[7] H. Mühlenbein. Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization. In *Proceedings of the Third International Conference on Genetic Algorithms, ICGA-89*, pp. 428–433, Georg Mason University, Morgan Kaufmann Publishers, 4–7 June 1989.

[8] H. Mühlenbein. New Solutions of the Mapping of Parallel Systems – Evolution Approach. *Parallel Computing* 4:269–279, 1987.

[9] H. Mühlenbein, M. Schomisch and J. Born. The Parallel Genetic Algorithm as Function Optimizer. *Parallel Computing* 17:619–632, 1991.

[10] M. Gorges-Schleuter. ASPARAGOS – An Asynchronous Parallel Genetic Optimization Strategy. In *Proceedings of the Third International Conference on Genetic Algorithms, ICGA-89*, pp. 428–433, Georg Mason University, Morgan Kaufmann Publishers, 4–7 June 1989.

[11] I. De Falco, R. Del Balio, E. Tarantino, R. Vaccaro. Simulation of Genetic Algorithms on MIMD Multicomputers. *Parallel Processing Letters* 2(4):381–389, 1992.

[12] R. Bianchini and C. Brown. Parallel Genetic Algorithms on Distributed-Memory Architectures. *Proceedings of the Sixth Conference on the North American Transputer Users Group, Transputer Research and Applications NATUG-6*, pp. 67–82, IOS Press, Amsterdam, The Netherlands, 1993.

[13] M. Schwem. Implementation of Genetic Algorithms on Various Interconnection Networks. In *Proceedings of Parallel Computing and Transputer Applications, CIMNE* 1:195–203, Barcelona, Spain, 1992.

[14] Thinking Machines Corporation, Inc. *Connection Machine CM-5 Technical Summary*. Cambridge, Massachusetts, USA, November 1992.

[15] U. Kohlmorgen. Parallel Genetic Algorithms. In J. Alander, Ed., *Proceedings of the First Nordic Workshop on Genetic Algorithms and Their Applications, 1NWGA-95*, pp. 135–143, University of Vaasa, Finland, January 1995.

[16] J. Saarinen, K. Kaski and J. Viitanen. A Parallel Multiprocessor System for Monte Carlo Simulations in Statistical Physics. *Review of Scientific Instruments* 60(9):2981–2991, September 1989.

[17] J. Saarinen, J. Tomberg, L. Vehmanen and K. Kaski. VLSI Implementation of Tausworthe Random Number Generator for Parallel Processing Environment. *IEE Proceedings-E* 138(3):138–146, May 1991.

[18] A. DeCegama. *The Technology of Parallel Processing, Parallel Processing Architectures and VLSI Hardware*, Vol. 1, Prentice Hall, 1989.

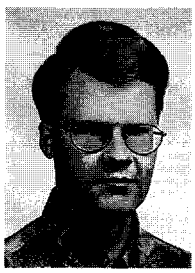
[19] Jong-man Park, Jeon-gue Park, Chong-hyun Lee and Munsung Han. Robust and Efficient Genetic Crossover Operator: Homologous Recombination. In *Proceedings of 1993 International Joint Conference on Neural Networks, IJCNN-93* 3:2975–2978, Nagoya, Japan, 25–29 October 1993.

[20] P. Kotilainen, J. Saarinen and K. Kaski, Neural Network Computation in a Parallel Multiprosessor Architecture. In *Proceedings of 1993 International Joint Conference on Neural Networks, IJCNN-93* 2: 1979–1982, Nagoya, Japan, October 25–29 1993.

[21] T. Hämäläinen, J. Saarinen and K. Kaski. TUTNC: A General Purpose Parallel Computer for Neural Network Computations. *Microprocessors and Microsystems* 19(8):447–465, 1995.

[22] Texas Instruments Inc. *Second Generation TMS320 User's Guide*. 1989.

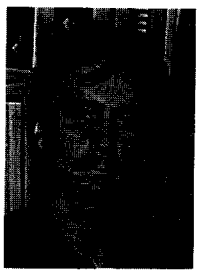
[23] Xilinx Inc. *The XACT4000 Data Book*. CA, USA, 1994.



Timo Hämäläinen was born in Finland on June 13, 1968. He studied analog and digital electronics, computer architecture and power electronics in the Electrical Engineering Department at Tampere University of Technology where he received the M.Sc. degree in 1993. He is currently working as research scientist in electronics laboratory at TUT. His Ph.D. research concerns on parallel processing of adaptive, intelligent algorithms in a special multiprocessor computer.

Harri Klapuri was born in Finland on October 4, 1970. He studied software systems, computer engineering and mathematics in the Information Engineering Department at Tampere University of Technology where he received the M.Sc. degree in 1995. He is currently working in electronics laboratory at TUT. His research interests include parallel processing, compilers and programming languages.

Jukka Saarinen was born in Finland on July 11, 1961. He studied computer architecture, digital techniques, telecommunications and software engineering in the Electrical Engineering Department at Tampere University of Technology where he received the M.Sc. degree in 1986, the Licentiate in Technology degree in 1989, the Doctor of Technology degree in 1991. Currently he is the research manager of neural networks research group in Electronics Laboratory at Tampere University of Technology. His research interest are parallel processing, neural networks, fuzzy logic and pattern recognition.



Pekka Ojala was born in Finland on May 21, 1964. He received the M.Sc. degree in 1988 and Licintiate in Technology in 1991 from the department Electrical Engineering at Tampere University of Technology. He also received M.S.E.E. Degree in 1991 from the department of Electrical Engineering at Brown University, Providence,, R.I. He finished the Ph.D. degree in 1994 at Tampere University of Technology, Microelectronics Laboratory. His research interest are modelling of MOSFET's and GaAs MESFET's as well as neural networks and gentic algorithms. He is currently working at Sierra Semiconductor, Inc.

Kimmo Kaski was born in Finland on April 20, 1950. He received the M.Sc. degree in 1973 and the Licentiate in Technology degree in 1977 from the Department of Electrical Engineering at Helsinki University of Technology, Finland. He finished the Ph.D. degree in the Theoretical Physics Department at Oxford University in 1981. Currently he is the Professor of Microelectronics at Tampere University of Technology, Tampere, Finland. He is acting as research leader in parallel processing hardware and software, neural network modeling, and computational physics areas.