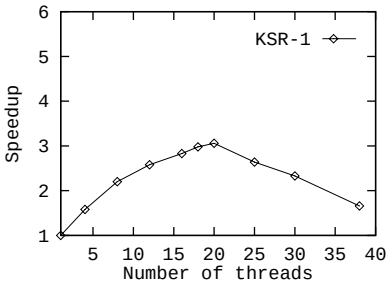

Fig. 3. Speedup vs. nr. of processors on the KSR1.

4. T. Fogarty, Implementing the Genetic Algorithm on Transputer Based Parallel Processing Systems, Parallel Problem Solving from Nature 1 (1991) 145–149
5. J.P. Cohoon, W.N. Martin, D.S. Richards, A Multi–population Genetic Algorithm for Solving the K–Partition Problem on Hyper–cubes, Proc. 4th Intl. Conf on Genetic Algorithms (1991) 244–248
6. R.J. Collins, D.R. Jefferson, Selection in Massively Parallel Genetic Algorithms, Proc. 4th Intl. Conf. on Genetic Algorithms (1991) 249–256
7. P. Spiessens, B. Manderick, A Massively Parallel Genetic Algorithm, Proc. 4th Intl. Conf. on Genetic Algorithms (1991) 279–285
8. C.C. Pettey, M.R. Leuze, A Theoretical Investigation of a Parallel Genetic Algorithm, Proc. 3rd Intl. Conf. on Genetic Algorithms (1989) 398–405
9. R. Tanese, Distributed Genetic Algorithms, Proc. 3rd Intl. Conf. on Genetic Algorithms (1989) 434–439
10. M.G.A. Verhoeven, E.H.L. Aarts, E. v. de Sluis, Parallel Local Search and the Travelling Salesman Problem, Parallel Problem Solving from Nature 2 (1992) 543–552
11. T. Maruyma, A. Konagaya, l. Konishi: An Asynchronous Fine–Grained Parallel Genetic Algorithm, Parallel Problem Solving from Nature 2 (1992) 563–572
12. H. Tamaki, Y. Nishikawa: A Parallel Genetic Algorithm based on a Neighborhood Model and Its Application to the Jobshop Scheduling, Parallel Problem Solving from Nature 2 (1992) 573–582
13. H. Mühlenbein, Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization; J.D. Becker, I. Eisele, F.W. Mündemann (Eds.): Parallelism, Learning, Evolution, Lect. Notes in Comp. Sci. 565, (Springer,Berlin, 1991) 398–406
14. R. Hauser, H. Horner, R. Männer, M. Makhaniok, Architectural Considerations for NERV—a General Purpose Neural Network Simulation System; in J. D. Becker, I. Eisele, F. W. Mündemann (Eds.): Parallelism, Learning, Evolution, Lect. Notes in Comp. Sci. 565, (Springer, Berlin, 1991) 183–195
15. The VMEbus Specification, Rev. C, VMEbus Int'l Trade Association (1987)
16. R.M. Stallman, Using and Porting GNU CC, Free Software Foundation (1992)
17. R. Schuhmacher (Ed.), One Year KSR1 at the Univerity of Mannheim: Results and Experience, RUM 35/93, University of Mannheim (1993)
18. H. Kredel, Computeralgebra on a KSR1 Parallel Computer, in [17], 26–34

This article was processed using the LaTeX macro package with LLNCS style

does not assign a thread to a specific processor but reschedules them every time they are runnable the situation may get even worse since they have to load their working set from the memory again.

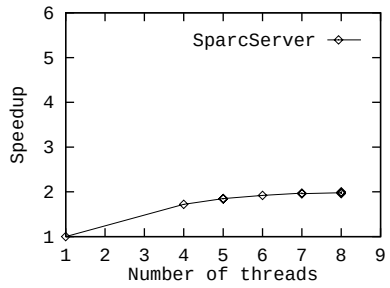

Fig. 2. Speedup vs. nr. of processors on the SparcServer system.

KSR1. The KSR1 allows for more user control over processors. The system is usually configured in a number of partitions with a given number of processors. The program can be run in one such partition and allocate all available processors. The largest usable partition contained 20 processors. The run time decreases as the number of threads is increased as long as we have less threads than processors. Then we can see that the run time starts to increase again. It seems the system overhead for scheduling the threads becomes significantly large at this point.

One major drawback of the KSR1 is the large increase in access time for the different stages of the memory hierarchy. The first-level cache (256 kB) needs two clock cycles, the 32 MByte local cache needs 20 clock cycles and an access to a remote cache needs 140 clock cycles. This is nearly 2 orders of magnitude and our application definitely does not fit into the first-level cache.

Several experience reports [17] about the KSR1 show, that one can achieve a significant speedup if the problem is carefully adapted to the machine specific parameters. However our results are more in line with the report on the port of an existing application [18] whose speedup usually did not exceed a factor of 5.

References

1. J.H. Holland, Adaption in Natural and Artificial Systems (The University of Michigan Press, Ann Arbor, 1975)
2. D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, (Addison–Wesley, Reading, 1988)
3. M. Gorges–Schleuter, ASPARAGOS: An Asynchronous Parallel Genetic Optimization Strategy, Proc. 3rd Intl. Conf. on Genetic Algorithms (1989) 422–427

For a C program the preferred method of parallelizing a task is to use the POSIX threads library (Pthreads). The functionality is essentially the same as with the Solaris threads library. Therefore the main GA program will be exactly the same, only the *threads_init()* and *threads_sync()* routines had to be adapted.

8 Results

The figures show the speedups which can be achieved by the different systems depending on the number of processors or threads used.

NERV. For the NERV system the programmer has complete control of the system and can decide how many processors he wants to use. The overhead is only marginal, since the necessary functions are directly supported by the hardware. If the number of processors is increased the communication time stays constant. The common bus however sets an upper limit to the extensibility of the sytem—it is not reasonable to consider a system with more than 40 processors in a VME–crate. The speedup is linear although below the theoretical maximum.

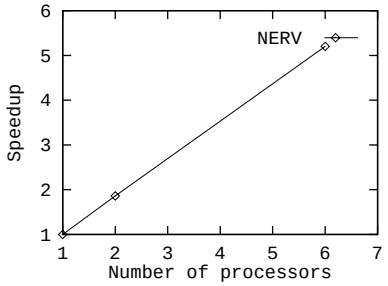

Fig. 1. Speedup vs. nr. of processors on the NERV system.

SparcServer. The SparcServer and the KSR1 are both very similar in that they provide a global shared memory and allow parallelization via a threads package. However, on the Sparc machine the user has no control over the processor resources. He can not specify on how many processors his program will run but must leave this decision to the operating system. The results show that the speedup is only in the order of two for an eight processor system. The system was in multiuser mode although no other computing intensive task was running during the measurements. It is difficult to give a specific reason for this behaviour. The working set of the program is quite large since several hundred net lists must referenced in each fitness evaluation. Therefore the local cache of each processor will usually not suffice to hold all relevant data. If the scheduler

This is all communication which will occur. All other values are fetched from local memory. A broadcast facility is the most efficient way to implement this since it does not depend on the number of processors. If we increase the number of processing elements we will decrease the time needed for each step while the communication overhead will stay constant.

6 SparcServer

The SparcServer 2000 is a commercially available shared memory system with up to 16 processors which supports symmetric multiprocessing. All processors have access to a global shared memory. The system runs the Solaris operating system which is responsible for load-balancing.

The normal way to take advantage of the multiple processors in the system is to use the *threads*-library. For synchronizing access to critical regions there are a number of mechanisms like *mutex* and *condition* variables. The thread library is very similar to the POSIX threads interface, although not completely identical.

The same arguments we used for the NERV–system apply here as well. Most parts of the algorithm can be parallelized ideally, but we need a synchronization after each major step. As long as we follow the same programming style as in the NERV–system it is unnecessary to lock data structures on a lower level, since there are no concurrent writes by different threads into the same memory area.

The implementation strategy is therefore to start N worker threads where each one is working on part of the population. The main thread is only responsible for initialization and controlling the synchronization of the other threads.

The synchronization has been implemented on top of the threads library. The GA routine uses only two functions, *thread_init()* and *thread_sync()*. The initialization routine is called once at the beginning of the program. It starts the worker threads and initializes the global mutex and condition variables. The number of worker threads can be given as a program argument. This allows us to vary the maximal number of processors.

7 KSR1

The KSR1 from Kendall Systems Research has some features which distinguish it from the more convential shared memory systems like the SparcServer. Although it looks like a global shared memory system from the programmer's point of view, there is no main memory in the usual sense at all. Instead each processor has a large cache memory of 32 MByte which is backed by mass storage and kept consistent by a cache coherence protocol. The interconnection network is invisible to the programmer, although the latency of memory updates may vary if two distant nodes have to communicate.

Each processor in the system is running an OSF/1 kernel, providing the usual Unix services. The machine can be split in several partitions with a certain number of processors dedicated to a certain program.

Crossover. As already mentioned we decided to generate the next generation by looping over all individuals of the new population and either copying an individual from the old one or create a new one by crossover from two parents. Again each processor will be responsible for a part of the population:

```
for ( i =  "first individual"; i <= "last individual"; i++) {
  offspring = &newPopulation[i];
  parent1 = select();
  parent2 = random_select();
  if (random(CROSSOVER_PROB) < CROSSOVER_PROB) {
    k = random(CHROM_LENGTH);
    for(j = 0; j < k; j++)
      offspring[j] = parent1[j];
    for(j = k; j < CHROM_LENGTH; j++)
       offspring[j] = parent2[j];
  } else /* copy individual */
    for(j = 0; j < CHROM_LENGTH; j++)
      offspring[j] = parent1[j];
}
synchronize();
```

After this step $P \cdot L$ elements will have been broadcasted (assuming that we encode e.g. each bit in a separate character) and each processing element will have a complete copy of the new population.

Mutation. The mutation operator is parallelized in the same fashion as the other operators. Again each processor handles $\frac{P}{N}$ chromosomes and broadcasts the results.

```
for(i = "first individual"; i <= "last individual"; i++) {
  individual = &newPopulation[i];
  for(j = 0; j < CHROM_LENGTH; j++)
    if (random(MUTATE_PROB) < MUTATE_PROB)
      individual[j] = !individual[j];
}
synchronize();
```

The broadcast of a bit changed by mutation is done by the assignment to *individual[j]*. Note that the right hand side of this assignment will only access local memory since it is a read access.

5.3 Discussion of NERV implementation

The program will transfer P fitness values (from step 1) and $P \cdot L$ bits for the new population (from step 2) over the common bus. In addition it must transfer the bits which are changed during mutation which may vary in each generation.

other processors. A read from this region will simply return the data in the local memory. For software written in C or C++ a programmer might take advantage of this property in the following way:

A pointer - either to a global variable or dynamically allocated - can be modified in such a way that it points into the broadcast region by a special function called *mk_global()*. Whenever this pointer is dereferenced by a write access an implicit broadcast happens. A read access will return the local data.

Note however that there is no explicit synchronization between the processors. If two processors update the same element, the last one will win.

A second extension on the VMEbus includes a hardware synchronization of all processing elements. The programmer calls a procedure *synchronize()* which will only return after all processors have reached the synchronization point.

5.2 Implementation of the Parallel Genetic Algorithm.

The previous sections suggest the following setup for the algorithm on the NERV system:

The same program is loaded onto each processor. Every processor has a copy of all individuals in his local memory. The current population and the population of the next generation are accessed by two pointers which have been prepared so that they both point into the broadcast region. The same holds for an array which contains the fitness values of all individuals. After each generation the two population pointers are simply exchanged. Let N be the number of processing elements in the system. The general strategy will be to distribute the computational load equally among all processing elements by assigning $\frac{P}{N}$ individuals to each processor.

The parallelization of each GA operator is now straightforward.

Fitness evaluation. Each processor evaluates the fitness of the individuals it has been assigned. The fitness values are simply written into the mentioned array which will automatically initiate a broadcast. Since each processor is responsible for another set of individuals no overlap will occur.

```
int fitness_values[POP_SIZE];
int *fitness;

fitness = mk_global(fitness_values);
for ( i = "first individual"; i <= "last individual", i++)
  fitness[i] = eval(i);
synchronize();
```

Note that the evaluation function uses only the local copy of the population. The access to *fitness[i]* is the (implicit) broadcast.

4 The Problem

In the following we present some results of such a parallelization on a number of different multiprocessor systems. We will show that only a small number of properties are required to get an efficient parallel program which implements the standard GA. The systems are all of the MIMD type but range from a special purpose system (NERV) to global shared memory systems (SparcServer, KSR1).

We implemented the same program on all machines. Instead of using some kind of toy problem we decided to use a real application as a benchmark. The task of the GA is to optimize the placement of logic cells in a field programmable gate array (Xilinx). The input is a design file, which is usually created by an external program, as well as a library of user–defined parts and information about the specific chip layout and package. From this input the program creates an internal list of the required logic cells and their connections.

Our test design used 276 logic blocks out of 320 possible on a Xilinx 3090 and 121 I/O blocks. The number of internal connections is in the order of several thousands. Since the chromosomes represent positions for each logic or I/O block, the chromosome length is given by the sum of these two numbers.

The program is completely CPU–bound until it writes the final output file. The big advantage of having a real application is that it does not have the usual problems of benchmarks, like being so small that they fit in the cache of the processor or concentrating the whole computational task in a few lines like in a matrix multiplication. Furthermore we have a simple criterion for comparing the implementations: how fast is the program speeded up by using multiple processors. Since the user may wait for the result of the placement program, measuring the real time to do the task seems to be the most reasonable solution.

5 The NERV multiprocessor system

5.1 The Hardware.

The NERV multiprocessor [14] is a system which has been originally designed for the efficient simulation of neural networks. It is based on a standard VMEbus system [15] which has been extended to support several special functions. Each processing element consists of a MC68020 processor with static local memory (currently 512 kB) and each VME board contains several processor boards. Usually the system is run in a SPMD (Singe Program Multiple Data) style mode, which means that the same program is downloaded to each processing element, while the data to be processed are distributed among the boards.

The following extensions to the VMEbus have been implemented in the NERV system:

A broadcast facility which is not part of the standard VME protocol. It allows each processor to access the memory of all other processor boards with a single write cycle. From the programmer's point of view there exists a region in his address space, where a write access will initiate an implicit broadcast to all

Mutation The mutation operation can be applied to each bit of each individual independently. Besides from the bit value the only information needed is the global parameter P_M.

3.2 Parallelization

It should be noted that it is usually not possible to gain a larger speedup for steps 1) and 2) because of data dependencies between the different steps of the algorithm. This can be seen e.g. for step 2: If the crossover operation selects one of the parents it does this according to its relative fitness. However this can only be done if the fitness values of all other individuals are already computed so that the mean value is available.

In the following we will point out what kind of data each processing element must access to perform the different steps of the algorithm.

Fitness evaluation. Each processing element must have access only to those individuals whose fitness it will compute. In the optimal case (number of processing elements = number of individuals) this is one individual. However the result of this computation is needed by all other processing elements since it is used for computing the mean value of all function evaluations needed in step 2.

Crossover. Each processing element which creates a new individual must have access to all other individuals since each one may be selected as a parent. To make this selection the procedure needs all fitness values from step 1.

Mutation. As in step 1 each processing element needs only the individual(s) it deals with. As mentioned above the parallelization could be even more fine grained as in steps 1 and 2, in which case each processing element would need only one bit of each individual. This could usually only be achieved by a SIMD style machine.

Many implementors of parallel genetic algorithms have decided to change the standard algorithm in several ways.

The most popular approach is the partitioning of the population into several subpopulations [5,9] or introducing a topology, so that individuals can only interact with nearby chromosomes in their neighborhood [3,6,10,12,13]. All these methods obviously reduce the coupling between different processing elements.

Although some authors report improvements in the performance of their algorithm and the method can be justified by biological reasons, we consider it as a drawback that not the original standard GA could be efficiently implemented. The reason is that a genetic algorithm is often a computational intensive task. It often depends critically on the given parameters used for the simulation (e.g. P_M and P_C). There are some theoretical results about how to choose these parameters or the representation of a given problem, but most of them deal with the standard GA only. Even then one often has to try several possibilities to adjust the parameters optimally.

$$h_i \propto \frac{f_i}{\bar{f}} \tag{1}$$

where f_i is the fitness of individual i and $\bar{f}$ the average over all fitness values.

Crossover The crossover operator takes two individuals from the population and combines them to a new one. The most general form is uniform crossover from which the so called one–point crossover and two–point crossover can be derived. First two individuals are selected, the first one according to its fitness and the second one by random. Then a crossover mask M_i, $i = 1, \ldots, L$, where L is the length of the chromosome, is generated randomly. A new individual is generated which takes its value at position i from the first individual if $M_i = 1$ and from the second one if $M_i = 0$. The crossover operator is applied with probability P_C.

Mutation Each bit of an individual is changed (e.g. inverted) with probability P_M.

All three steps above are iterated for a given number of generations (or until one can no longer expect a better solution).

3 Parallel Genetic Algorithms

It has long been noted that genetic algorithms are well suited for parallel execution. Parallel implementations of GAs exist for a variety of multiprocessor systems, including MIMD machines with global shared memory [11] and message passing systems like transputers [3,4] and hypercubes [8] as well as SIMD architectures [6,7] like the Connection Machine.

3.1 Dependencies

It is easy to see that the following steps in the algorithm can be trivially parallelized:

Evaluation of fitness function The fitness of each individual can be computed independently from all others. This gives us a linear speedup with the number of processing elements.

Crossover. If we choose to generate each individual of the next generation by applying the crossover operator, we can do this operation in parallel for each new individual. The alternative would be to apply crossover and to put the resulting individual in the existing population where it replaces e.g. a bad solution.

Implementation of Standard Genetic Algorithm on MIMD machines *

R. Hauser[1], R. Männer[2]

[1] CERN, Geneva, Switzerland
email: rhauser@cernvm.cern.ch
[2] Lehrstuhl für Informatik V, Universität Mannheim, Germany
email: maenner@mp-sun1.informatik.uni-mannheim.de

Abstract. Genetic Algorithms (GAs) have been implemented on a number of multiprocessor machines. In many cases the GA has been adapted to the hardware structure of the system. This paper describes the implementation of a standard genetic algorithm on several MIMD multiprocessor systems. It discusses the data dependencies of the different parts of the algorithm and the changes necessary to adapt the serial version to the parallel versions. Timing measurements and speedups are given for a common problem implemented on all machines.

1 Introduction

In this paper we describe the implementation of a standard Genetic Algorithm [1,2] on a number of different multiprocessor machines. After the discussion of the data dependencies in a straight forward implementation of a GA, the parallelization strategy on each machine is discussed and speedup measurements are presented. No attempt is made to optimize the algorithm for each specific architecture, instead we tried to stay as closely as possible to the original and parallelize in a way which seems natural for an application programmer on such a machine. This includes the use of standard libraries like *Pthreads* if available and e.g. recommended by the vendor as the preferred method of parallelization.

2 Genetic Algorithms

In this section we shortly outline the GA we have implemented on the different machines.

Selection The fitness of each individual in the population is evaluated. The fitness of each individual relative to the mean value of all other individuals gives the probability with which this individual is reproduced in the next generation. Therefore the frequency h_i of an individual in the next generation is given by

* This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant Ma 1150/8–1.

REIHE INFORMATIK
8/95

Implementation of Standard Genetic Algorithm on MIMD machines

R. Hauser, R. Männer
Universität Mannheim
Seminargebäude A5
D-68131 Mannheim