Proceedings of the 1994 ACM/SIGAPP Symposium on Applied Computing March 6-8, 1994, pp.483-487, ACM Press.

A PARALLEL ISLAND MODEL GENETIC ALGORITHM FOR THE MULTIPROCESSOR SCHEDULING PROBLEM*

Arthur L. Corcoran **Roger L. Wainwright**

The University of Tulsa

Keywords: *Parallel Processing, Multiprocessor Scheduling, Genetic Algorithms, Parallel Island Model.*

Abstract

In this paper we compare the performance of a serial and a parallel island model Genetic Algorithm for solving the Multiprocessor Scheduling Problem. We show results using fixed and scaled problems both using and not using migration. We have found that in addition to providing a speedup through the use of parallel processing, the parallel island model GA with migration finds better quality solutions than the serial GA.

1 Introduction

The advent of time sharing systems signaled the beginning of efforts to maximize the use of processor resources through the use of concurrent and parallel processing. Simultaneous and concurrent execution of tasks meant that CPU utilization could be greatly increased and higher throughput could be obtained. However, multitasking and multiprocessing systems have introduced a new problem with respect to scheduling these tasks. While simple to express, this scheduling problem has proven intractable.

The quest for efficient, near optimal solutions to this and other combinatorial problems has led to the development of Genetic Algorithms (GAs). GAs borrow ideas from natural evolution to efficiently search a problem domain and effectively find near optimal solutions in a short amount of time. Parallel GAs have been developed to speed up this process as well as obtain higher quality solutions.

This paper is organized as follows: Section 2 introduces the Multiprocessor Scheduling Problem, including examples, related problems, and summarizing recent work. Section 3 provides an introduction to Genetic Algorithms, illustrating how they work and including an overview of the basic concepts. Section 4 gives a brief overview of the different forms of parallelism used in GAs and of the island model in particular. References to recent work in parallel GAs are provided. Section 5 summarizes our experimental results comparing serial and parallel island model GAs on the Multiprocessor Scheduling Problem. Finally, Section 6 provides an analysis of our experimental results.

*Research partially supported by OCAST Grant AR2-004 and Sun Microsystems, Inc.

2 Multiprocessor Scheduling

The multiprocessor scheduling problem [4] is defined as follows: a set of n jobs is to be scheduled on a set of m identical processors. A schedule is simply the sequence in which the jobs are executed. Each job J is specified as $J = (t, c)$, where c is the capacity (memory) requirement of the job and t is its running time. Note that only nonpreemptive job scheduling is considered. That is, once a job is started it remains in the processor until it is finished. It is assumed that the additional time to run multiple tasks on a single processor is negligible. The objective is to determine a schedule of jobs on the machines so as to minimize the total processing time. For example, consider the case where $m = 3$, each processor has memory capacity $c = 5$, and there are $n = 12$ jobs to be scheduled:

Job	1	2	3	4	5	6
	(1,2)	(2,2)	(2,1)	(2,4)	(3,1)	(1,3)

Job	7	8	9	10	11	12
	(2,2)	(2,3)	(4,2)	(3,4)	(2,1)	(2,3)

An example schedule is shown in Figure 1 requiring 5 time units. An optimal schedule requiring 4 time units is shown in Figure 2.

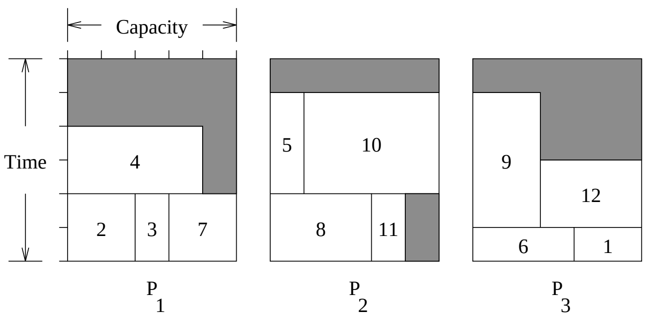

Figure 1: An Example Schedule

Note the Job Shop Scheduling problem is a variation of the multiprocessor scheduling problem. In the job shop scheduling problem each job, J_i requires the completion of several tasks, $T_{j,1}$, $T_{j,2}$, ..., $T_{j,n}$. The tasks for any job J_i are to be carried out in the order 1, 2, 3, ..., etc., where each task j cannot begin until task $j-1$ $(j > 1)$ has been completed [8]. In the job shop scheduling problem, the processor capacity is not considered. It is assumed every task in every job uses the entire memory capacity of a given processor.

The multiprocessor scheduling problem is also similar to the

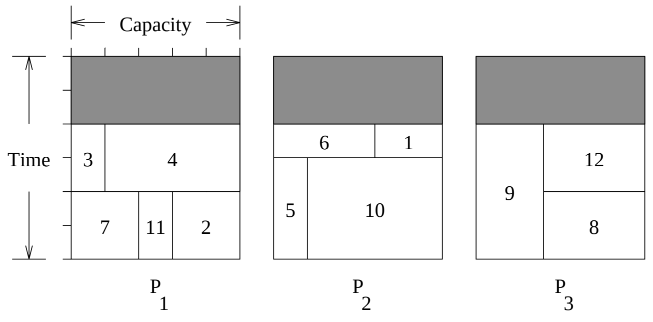

Figure 2: An Optimal Schedule

bin packing problem, where each processor is a bin, and each job is a package. Similarly, in the vehicle routing problem each vehicle is the same as a processor, and the capacity of goods received by each customer corresponds to a job. The multiprocessor scheduling problem, job shop scheduling, flow shop scheduling problem, bin packing and vehicle routing problems and all of their variations are part of a broad class of problems called the partition problem. The partition problem is a well known problem which has been shown to be NP-complete [4]. Note, when the multiprocessor scheduling problem and its variants are restricted to a fixed m, it is possible to devise a pseudo-polynomial time algorithm. However, in this paper we consider only the general problem where m is not fixed.

The multiprocessor scheduling problem and its variations of other scheduling problems are economically very important problems, especially in industrial applications. However there is no known solution to any of these problems that can be done in polynomial time. Consequently, researchers have concentrated on developing algorithms that search through the vast state-space of the problem in an efficient manner looking for *near optimal* solutions.

Yamada and Nakano [15] developed a GA implementation for large-scale job shop problems. Also, Davidor *et al.* [2] investigated GAs as a technique for solving the job shop scheduling problem. Kidwell [9] developed a GA to schedule distributed tasks on a bus-based system. Li and Cheng [11] developed a job shop scheduling algorithm to partition a mesh connected system, where jobs require square meshes and the system itself is a square mesh with size a power of two. This problem is similar to a two-dimensional bin packing problem. Corcoran and Wainwright [1] present a GA for solving the two dimensional bin packing problem.

3 Genetic Algorithms

A genetic algorithm is an iterative procedure which borrows the ideas of natural selection and 'survival of the fittest' from natural evolution. By simulating natural evolution, in this way, a GA can easily solve complex problems. Furthermore, by emulating biological selection and reproduction techniques, a GA can effectively search the problem domain in a general, representation-independent manner.

The genetic algorithm maintains a population or pool of candidate solutions for a given objective function. The candidate solutions represent an encoding of the problem into a form that is analogous to the chromosomes of biological systems. Each chromosome is made up of a string of genes (whose values are called alleles). The chromosome is typically represented in the GA as a string of bits. However, integers and floating point numbers can easily be used. Associated with each chromosome is a fitness value, which is found by evaluating the chromosome with the objective function. It is the fitness of a chromosome which determines its ability to survive and produce offspring.

Once an initial pool has been generated and all of its members have been evaluated, the genetic algorithm begins its emulation of the life cycle. The pool size remains constant throughout the GA. At each step in the iteration, chromosomes are probabilistically selected from the population for reproduction according to the principle of the 'survival of the fittest'. Offspring are generated through a process called crossover, which can be augmented by mutation. The offspring are then placed back in the pool, perhaps replacing other members of the pool. This process can be modeled using either a 'generational' [5, 7] or a 'steady-state' [14] genetic algorithm. The generational GA saves offspring in a temporary location until the end of a generation. At that time the offspring replace the entire current population. Conversely, the steady-state GA immediately places offspring back into the current population.

4 Parallel GAs

Parallel GAs (PGAs) can be classified according to three different models: global, island, and cellular. See Gordon and Whitley [6], and Knight and Wainwright [10] for more detail.

Global model PGAs use parallel techniques to speed up the operation of the GA without changing the basic operation of the sequential GA. In this model, the genetic algorithm employs a single global population without locality considerations. That is, a *panmictic* (global) mating strategy is used where mating probability is independent of location. This model requires global synchronization and control of the GA. Ironically, the global models have not received as much attention as the others because of machine dependency considerations.

Island models arose out of the desire to exploit coarse grained parallel architectures. While global models use a panmictic mating strategy, island models use a strategy usually found in a *polytypic* species, which evolve in isolated subgroups with more interaction within subgroups than between them. The island model typically runs a serial GA on each of several loosely connected processors. Each GA is identical to but independent of the others. Each GA is usually started with a different random seed. Periodically, individuals are transmitted between the islands in a process called *migration*. The island model strategy eliminates the global synchronization that is required in the panmictic approach. However, some synchronization is usually required for migration.

Cellular models (a term used by Whitley [13]) arose from the desire to exploit the fine grained, massively parallel architectures. These machines consist of a huge number of simple processors typically connected in a ring or torus topology. In general, the cellular model assigns one chromosome per processor and limits selection and mating to local neighborhoods or *demes*. While the island model has fixed boundaries between its subpopulations and a structured mechanism for migration, the cellular model has overlapping demes and incorporates migration to the same effect without the overhead involved. It achieves the same effect of isolation found in the island model by using isolation by distance. In this case, new species are formed at the boundaries between emergent structures. The cellular model also avoids the problems associated with the panmictic approach, such as global synchronization. However, cellular models require a large amount of communication.

5 Experimental Results

We developed all of our genetic algorithms using LibGA [1]. We implemented a typical serial GA for the multiprocessor scheduling problem, as well as a parallel GA using the island model. Since we used the same library to implement both GAs, the GA used on each processor in the island model was identical to the one used in the serial GA. The parallel GA was implemented using Intel's iPSC/2 simulator. Note, the serial GA was implemented without the simulator, however, it is identical to the parallel GA with one processor. All tests were run on a Sun SparcStation 10 Model 30 under Solaris 2.2. Processing times given are actual CPU time, which is independent of processor load. The parameters used by the GA include: generational reproduction with elitism, roulette selection, asexual crossover with probability 1.0, and swap mutation with probability 0.01. Asexual crossover and swap mutation are unique to LibGA [1]. Asexual crossover and swap mutation both swap two randomly selected genes. The crossover probability used was the default value used in LibGA. The remainder of the operators and parameters are those typically used in genetic algorithms [5].

We used four data sets in our tests: L25, R25, R50, and R75. The L25 data set is a contrived set composed of 25 tasks with a known optimal schedule time of 20. The R25, R50, and R75 data sets were generated at random and are composed of 25, 50, and 75 tasks, respectively. The optimal solutions for the R25, R50, and R75 data sets are unknown, however, they cannot be less than 23, 44, and 69, respectively.

Our goal in this research was to compare the performance of the serial and island parallel model GAs both in terms of time required to find a solution and the quality of the solution obtained. In order to be fair to both techniques, as well as to be thorough in our research, we compared the results for the serial GA to those for the parallel GA using both fixed and scaled problems. These results are summarized below.

5.1 Serial GA

Table 1 shows our results for the serial GA. Each set was run sixteen times using different initial seeds. Each run was limited to 500 generations and the pool size was 400. The columns in the table report the best result obtained for each set (x_{min}), as well as the average ($\bar{x}$), variance (σ^2), and standard deviation (σ) for the set of 16 runs. The rows in the table report statistics for each set, which are the fitness' obtained for each run (f_r), and the number of generations (m) and CPU time in seconds (T_1) when f_r first appeared in the population[1]. For example, results for the L25 set indicate the best performing run obtained a fitness of 20 (the optimum) in 29 generations or 27.66 seconds of CPU time. For the L25 data set on average, the fitness was 20.44, the number of generations was 77.88, and the average CPU time was 75.27 seconds.

Set		x_{min}	$\bar{x}$	σ^2	σ
L25	f_r	20	20.44	3.958e−01	6.292e−01
	m	29	77.88	8.224e+02	2.868e+01
	T_1	27.66	75.27	7.848e+02	2.801e+01
R25	f_r	24	24.69	3.625e−01	6.021e−01
	m	25	120.94	2.289e+04	1.513e+02
	T_1	23.60	118.60	2.257e+04	1.502e+02
R50	f_r	46	46.44	5.292e−01	7.274e−01
	m	161	287.44	8.977e+03	9.475e+01
	T_1	179.89	323.18	1.156e+04	1.075e+02
R75	f_r	73	74.06	7.292e−01	8.539e−01
	m	289	361.56	8.892e+03	9.430e+01
	T_1	344.71	434.57	1.301e+04	1.140e+02

Table 1: Results for Serial GA

5.2 Fixed PGA

Our fixed PGA maintained a total population size of 400, as used in the serial GA. This means that the population size on each processor, n, is determined by dividing the total population size, N, by the number of processors used, p. Thus, $n = N/p$. Clearly, as more processors are added to the fixed problem, each processor has a smaller pool to work with. This has the advantage of speeding up the execution of the GA and the disadvantage of reducing the GAs sampling efficiency. However, the fixed problem ensures the sampling of the search space examined is the same size for both the serial and parallel GAs.

Table 2 shows our fixed results without migration. Each data set was tested with 2, 4, 8, and 16 processors. For each set and number of processors (p), the table lists the final fitness value (f_r), the generation (m) and time (T_p) when f_r first appeared, and the percentage of processors containing f_r ($\%_r$). The table clearly indicates that the addition of more processors led to faster CPU times. However, this was at the expense of solution quality. In general, the sequential GA performed better in the quality of the solution than the fixed PGA without migration. Compare the f_r column in Table 2 with the x_{min} column and f_r rows in Table 1.

Table 3 shows our fixed results with migration. The table lists f_r, m, T_p, and $\%_r$, which are defined above. In addition, there are two new parameters: migration interval (M_i) and migration rate (M_r). The migration interval is the number of iterations between migrations. We used M_i values of 5, 25, 50, 75, and 100 iterations which respectively represent 1%, 5%, 10%, 15%, and 20% of the total number of iterations allowed. The migration rate is the percentage of the pool which will migrate. We used M_r values of 1%, 5%, 10%, 15%, and 20%. Migrants were selected with the GAs selection function and passed to one neighbor using a ring topology. Migrants accepted from the other neighbor replaced the worst M_r percent of the population.

Table 3 clearly has exceptional performance in the L25 and R25 data sets. The addition of more processors reduces execution time without sacrificing solution quality. The R50

[1] The time required to reach 500 generations is not included as it showed little change between runs and has no bearing on our discussion.

Set	p	f_r	m	T_p	$\%_r$
L25	2	20	70	17.07	50
	4	20	94	6.80	50
	8	21	51	1.27	25
	16	22	28	0.30	19
R25	2	25	33	7.95	100
	4	25	109	7.89	25
	8	24	144	3.58	13
	16	25	52	0.56	13
R50	2	48	96	30.73	100
	4	48	326	34.72	25
	8	48	120	4.33	13
	16	50	74	1.16	19
R75	2	76	201	71.34	100
	4	76	282	34.40	25
	8	80	83	3.89	25
	16	83	67	1.39	6

Table 2: Results for Fixed PGA without Migration

Set	p	f_r	m	T_p	M_i	M_r	$\%_r$
L25	2	20	55	13.36	5	5	100
	4	20	124	8.83	25	15	100
	8	20	54	1.36	5	15	100
	16	20	70	0.65	75	10	44
R25	2	24	453	112.03	25	1	100
	4	24	67	4.77	25	10	100
	8	24	34	0.85	25	1	100
	16	24	81	0.78	25	15	100
R50	2	46	307	97.98	25	10	100
	4	46	160	16.65	25	1	100
	8	46	267	9.12	100	15	50
	16	46	403	5.77	75	10	13
R75	2	73	471	164.24	75	20	50
	4	73	338	40.19	50	5	100
	8	74	311	13.84	25	15	100
	16	75	198	3.87	50	15	44

Table 3: Results for Fixed PGA with Migration

data set has similar results, yet the percentage of islands containing the best solution is beginning to dwindle. This is better seen in the R75 data set, where the solution quality has degraded for larger numbers of processors. The results suggest that there is a minimum threshold population to search space size ratio that must be maintained to ensure an efficient search of the space. With a chromosome length of 75 there are more than 10^{109} possible solutions. With 16 processors there are only 25 chromosomes in each processor. It appears that this is insufficient genetic material to work with for such a large search space. As expected, the PGA works much better with migration than without migration.

Figure 3 shows the migration overhead using 16 processors. These are actual experimental results obtained for a typical run. The figure shows percentage overhead as a function of migration rate for three different migration intervals: 5, 25, and 50. When M_i is 5, migration occurs the most frequently. As the solid line in Figure 3 shows, when M_r is increased from 1% to 20%, the migration overhead quickly grows from 4% to nearly 20%. Larger migration intervals cause migration to occur less frequently. Migration intervals of 25 and 50 are illustrated in the figure with dashed and dotted lines, respectively. These larger intervals lead to slower growth in migration overhead. In most of our runs, the migration overhead never exceeded 5%, and was well worth the effort.

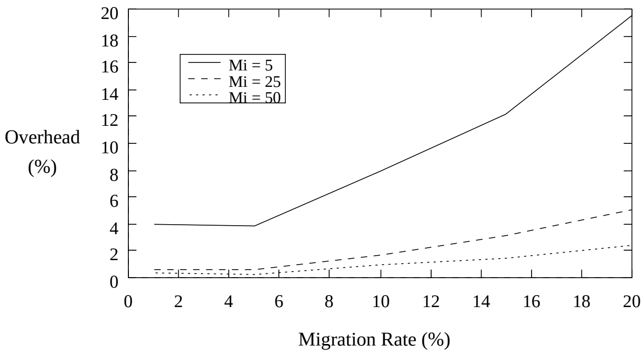

Figure 3: Migration Overhead with 16 Processors

5.3 Scaled PGA

Our scaled PGA used a population size of 400 *on each processor*. That is, the total population size increased as more processors were added. This model makes the maximum use of each available processor so that a larger sample of the search space can be examined in the same amount of time.

Table 4 shows our scaled results without migration. The columns p, f_r, m, T_p, and $\%_r$ are defined as before. Here we find that the time to find the f_r is longer compared to the fixed problem. However, this extra time was well worth the effort considering the improvement in solution quality. The addition of more processors had no effect on the solution quality, but in some cases reduced the time needed to find the solution. Note that the solutions found by the scaled PGA without migration were the same as the best obtained by the serial GA.

Set	p	f_r	m	T_p	$\%_r$
L25	2	20	95	91.68	50
	4	20	95	91.58	25
	8	20	95	91.52	13
	16	20	59	56.58	6
R25	2	24	83	80.11	50
	4	24	83	80.02	75
	8	24	83	79.96	38
	16	24	29	27.81	50
R50	2	46	205	197.86	50
	4	46	205	197.63	50
	8	46	205	197.49	38
	16	46	199	190.83	44
R75	2	73	414	399.58	50
	4	73	351	338.38	50
	8	73	351	338.14	25
	16	73	351	336.59	25

Table 4: Results for Scaled PGA without Migration

Table 5 shows our scaled results with migration. Column values are defined as before. For the first three data sets, we see that solution quality is the same as the fixed problem with migration. However, the times required to find the solution are greatly reduced. The solutions for these three sets match the best results from the serial GA. The times to obtain the solutions were somewhat higher than the best of the serial GA's, but were much better than the average serial GA time. The most impressive results are for the R75 data set. Here the scaled PGA with migration outperformed any other method. It was able to obtain 71 while the best the other methods could obtain was 73. This is especially remarkable considering these results are extremely close to optimal: for this data set the optimal can be no better than 69. Note the scaled PGA with migration actually has better solution quality as more processors are added. This is unlike the fixed PGA with migration which had worse solution quality as more processors were added. This is caused by reduced sampling efficiency from the use of smaller pool sizes for larger processor sets in the fixed PGA with migration.

Set	p	f_r	m	T_p	M_i	M_r	$\%_r$
L25	2	20	43	41.62	25	15	100
	4	20	35	34.14	5	1	100
	8	20	26	25.36	5	20	100
	16	20	27	25.53	25	20	100
R25	2	24	38	37.76	5	5	100
	4	24	46	44.78	5	5	100
	8	24	32	31.21	5	15	100
	16	24	30	28.37	5	10	100
R50	2	46	254	252.40	100	20	100
	4	46	115	111.95	25	10	100
	8	46	120	117.04	5	5	100
	16	46	105	116.79	25	15	100
R75	2	72	295	370.37	75	5	100
	4	72	311	376.10	100	5	100
	8	72	283	342.95	5	5	100
	16	71	496	591.50	25	1	6

Table 5: Results for Scaled PGA with Migration

6 Conclusions

In this paper, we have studied the multiprocessor scheduling problem and explored its implementation using a serial model and a parallel island model genetic algorithm. We found that the serial GA was able to find optimal or near optimal answers for the multiprocessor scheduling problem on our four data sets. In our experiments we tested a fixed and a scaled parallel island model GA both with and without migration.

We found that in both the fixed and scaled PGAs without migration we obtained roughly the same performance as the serial GA, except that performance tended to degrade in the fixed problem as more processors were added. This was caused by a reduction in sampling efficiency from the use of smaller pool sizes for larger processor sets. The inclusion of migration in the fixed PGA improved the performance for three of the four data sets. Performance started to decline in the fourth data set when the sample size per processor was smaller in proportion to the search space size than for the other data sets. In the scaled PGA without migration, we observed improved performance in solution quality with a small degradation in speed. However, this was to be expected, considering the increased sample size and corresponding increase in sampling efficiency. In the scaled PGA with migration, the increased sampling efficiency of a larger sample size combined with the collaborative like effects of migration resulted in a relatively large improvement in the solution quality.

Our results show the scaled PGA with migration offers both improved solution quality and speedup due to parallel processing. However, we found the fixed PGA offers speedup at the expense of solution quality. Because of this, we feel the scaled PGA with migration is far superior to the other methods tested. We expect this trend to continue for even larger problems than R75, as the search space size increases combinatorially.

In the future, we plan to examine this problem using a cellular PGA to determine if further improvement is possible. The cellular PGA is especially interesting because it more closely models the natural evolutionary process. It will also be useful to compare our results with other heuristic methods, such as hill climbing. In addition, we have recently received time on a Paragon and plan to run larger cases of our island PGA on that machine.

Acknowledgments

This research has been partially supported by OCAST Grant AR2-004. The authors also wish to acknowledge the support of Sun Microsystems, Inc.

References

[1] A. L. Corcoran and R. L. Wainwright. LibGA: A user-friendly workbench for order-based genetic algorithm research. In E. Deaton, K. M. George, H. Berghel, and G. Hedrick, editors, *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pages 111–118, New York, 1993. ACM Press.

[2] Y. Davidor, T. Yamada, and R. Nakano. The ECOlogical framework II: Improving GA performance at virtually zero cost. In Forrest [3].

[3] S. Forrest, editor. *Proceedings of the Fifth International Conference on Genetic Algorithms*, Urbana-Champaign, Illinois, July 1993. Morgan Kaufmann.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Massachusetts, 1989.

[6] V. S. Gordon and D. Whitley. Serial and parallel genetic algorithms as function optimizers. In Forrest [3].

[7] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[8] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1984.

[9] M. D. Kidwell. Using genetic algorithms to schedule distributed tasks on a bus-based system. In Forrest [3].

[10] L. Knight and R. Wainwright. HYPERGEN – a distributed genetic algorithm on a hypercube. In *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, Virginia, Apr. 1992.

[11] K. Li and K.-H. Cheng. Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme. *IEEE Transaactions on Parallel and Distributed Systems*, 2(4):413–422, Oct. 1991.

[12] R. Männer and B. Manderick, editors. *Parallel Problem Solving from Nature, 2.* North-Holland, Amsterdam, 1992.

[13] D. Whitley. Cellular genetic algorithms. In Forrest [3].

[14] D. Whitley and J. Kauth. GENITOR: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, Denver, Colorado, 1988.

[15] T. Yamada and R. Nakano. A genetic algorithm applicable to large-scale job-shop problems. In Männer and Manderick [12].