Genetic Algorithms for Parallel Code Optimization

Ender Özcan
Dept. of Computer Engineering
Yeditepe University
Kayışdağı, İstanbul, Turkey
Email: eozcan@cse.yeditepe.edu.tr

Esin Onbaşıoğlu
Dept. of Computer Engineering
Yeditepe University
Kayışdağı, İstanbul, Turkey
Email: esin@cse.yeditepe.edu.tr

***Abstract-* Determining the optimum data distribution, degree of parallelism and the communication structure on Distributed Memory machines for a given algorithm is not a straightforward task. Assuming that a parallel algorithm consists of consecutive stages, a Genetic Algorithm is proposed to find the best number of processors and the best data distribution method to be used for each stage of the parallel algorithm. Steady state genetic algorithm is compared with transgenerational genetic algorithm using different crossover operators. Performance is evaluated in terms of the total execution time of the program including communication and computation times. A computation intensive, a communication intensive and a mixed implementation are utilized in the experiments. The performance of GA provides satisfactory results for these illustrative examples.**

I. Introduction

Data distribution and processor allocation are two important factors that affect the performance of programs written for distributed memory parallel architectures. Distribution of the data among the processors affects the communication time. On the other hand, the number of processors used at each step of the parallel code (degree of parallelism) affects both the computation time and the communication time.

Different approaches have been used to solve the problem of optimizing data distribution in parallel programs [1]-[12]. These projects use a variety of optimization methods. There are also research works that present notation for communication-free distribution of arrays [13], [14].

The problem of finding optimal mappings of arrays for parallel computers is shown to be NP-complete [15]. As a discrete problem, even if a restricted set of data decomposition patterns is used, the nonlinear nature of the problem does not change, especially when combined with selecting the degrees of parallelism for each program stage and attempting to minimize the overall execution time. In this study, Genetic Algorithms (GA) is used to analyze the problem of determining the data distribution and the degree of parallelism for each stage of a parallel code in order to minimize the total execution time.

II. Performance Characterization Of Parallel Algorithms

A. Levels

A serial algorithm may be composed of a sequence of stages (denoted as levels), where each stage is either a single loop or a group of nested loops (Figure 1(a)). An example program segment is given in Figure 6, where there are five levels. The entire sequence of levels may also be enclosed by an iterative loop (Figure 1(b)). A general case is given in Figure 1(c), where some consecutive loops are enclosed by iterative loops, which again with adjacent loops may be enclosed by other iterative loops and so on, like a tree structure. Here, to simplify the case, it is assumed that programs are in the form of Figure 1(a) and Figure 1(b), and programs that have structure as in Figure 1(c) are reduced to the form in Figure 1(b) by combining L_2, L_3, L_4 as a single level.

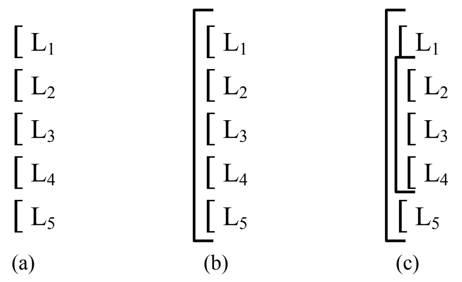


Figure 1. (a) sequence of levels, (b) sequence of levels enclosed by an iterative loop, (c) sequence of loops with a general structure

When the programs are parallelized, each level is assumed to have a different degree of parallelism that is, each level may be executed using a different number of processors. Levels requiring short computation may be parallelized on a few processors but those having long computations may require the use of many processors. Also, due to the distribution of the data among the processors, when the code is parallelized, communication may be required before the execution of each level. Although increasing the number of processors decreases the computation time of a level, it may cause extra communication between the levels.

B. Performance Evaluation

Performance of a serial algorithm can be expressed in terms of the problem size, but the performance of a parallel algorithm, in addition to the problem size, depends on the number of available processors, the distribution of the data among the processors, and the methods used for transferring data between processors.

At some stage l of a parallel code, execution time (t_{exec}) can be expressed as the sum of the computation and communication times

$$t^l_{exec} = t^l_{comp} + t^l_{comm} \qquad \text{Equation 1.,}$$

where t^l_{comp} denotes the computation time, and t^l_{comm} denotes the communication time at stage l.

Computation time can be formulated as

$$t^l_{comp} = t^l_{seq} / p_l \qquad \text{Equation 2.,}$$

where t^l_{seq} is the predicted computation time of stage l of the sequential algorithm on a single processor, and p_l is the number of processors used in the parallel code for stage l.

Communication time depends on the number of processors p_l and the communication structure c_l used for the transfer of data at that stage. t^l_{comm} consists of two terms, a term that increases with the size of data to be transferred, represented by f_1, and an overhead, represented by f_2,

$$t^l_{comm} = f_1(c_l, p_l)d_l + f_2(c_l, p_l) \qquad \text{Equation 3.,}$$

where d_l is the data size per processor. Note that both f_1 and f_2 are in the form (a p_l + b), where a and b are constants that depend on the communication structure.

The total execution time (T) of the program is the sum of the execution times of all stages in the program

$$T = \sum_{l=1}^{L} t^l_{exec} \qquad \text{Equation 4.,}$$

where L is the total number of stages (denoted as levels) in the code.

In order to determine the execution time of a parallel program at compile-time, machine characterization and performance prediction method given in [16] is used. In this method, computation and communication characteristics of the machine are measured and formulated. In order to calculate the total execution time of a parallel program, its program parameters (i.e. computation time, data size) are substituted in the formulas.

C. Data Decomposition and Alignment

In most of the parallel algorithms, arrays of one or higher dimensions are used. In this study, arrays are assumed to be distributed to the processors in the form of blocks. Block decompositions of 1 and 2-dimensional arrays on four processors are illustrated in Figure 2 and Figure 3, respectively. It is assumed that all scalar values are replicated on all processors.

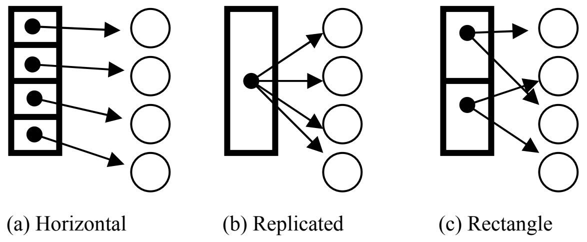


Figure 2. Possible decomposition patterns for a 1D array on 4 processors

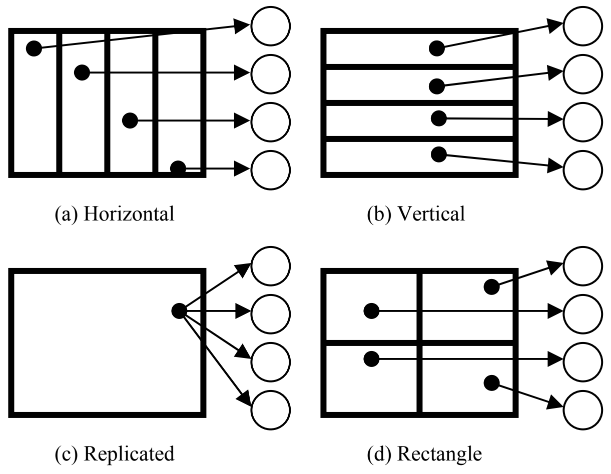


Figure 3. Possible decomposition patterns for a 2D array on 4 processors

Arrays are distributed to the processors according to one of the decomposition patterns, presented in Figure 2 and Figure 3. Due to the computational requirements, arrays existing at a level, may be distributed to processors using the same or different decomposition patterns. Considering all the arrangements of decomposition patterns for all the arrays at a level, some of them may not be feasible. Feasible arrangements of decomposition patterns of arrays at a level are referred as *alignment*.

As an example, possible alignments for the third level of the code in Figure 7 are demonstrated in Table I. The arrangement, (Horizontal, Vertical, Horizontal), for arrays c, I and b is not a feasible arrangement, as it can not satisfy the computational requirements. Hence, it is not accepted as an alignment.

TABLE I. ALLIGNMENTS FOR THE THIRD LEVEL OF THE CODE IN FIGURE 7

Array	Alignment 1	Alignment 2	Alignment 3
c	Horizontal	Vertical	Rectangular
I	Horizontal	Vertical	Rectangular
b	Horizontal	Vertical	Rectangular

D. Communication Structures

In distributed memory architectures using message-passing, generally, data is transferred among the processors in a structured way. Different communication structures have been defined for data exchange between processors [17], [18]. In this study, multiphase (MU), shift (SH), broadcast (BR), scatter (SC) and gather (GA) structures have been utilized (Figure 4).

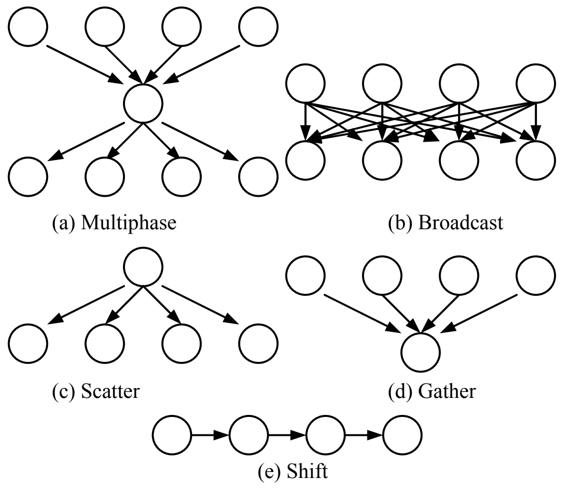


Figure 4. Five types of communication structures used in the study

Performance of the communication structures can be characterized in terms of f_1 and f_2, as shown in Table II. For all structures, f_1 and f_2 depend on the number of processors, except for SH, where f_1 and f_2 are constant.

The machine parameters M, N, R and Q are measured running benchmarks on the parallel machine, as explained in [16]. The hardware platform used in this study consists of 16 Pentium 4, 2GHz processors connected by 100Mbit/s interconnection network.

TABLE II. Machine Parameters For The Communication Structures

c	$f_1(c_l, p_l)$	$f_2(c_l, p_l)$
MU	$M_1\ p_l + N_1$	$R_1\ p_l + Q_1$
BR	$M_3\ p_l + N_3$	$R_3\ p_l + Q_3$
SC	$M_4\ p_l + N_4$	$R_4\ p_l + Q_4$
GA	$M_5\ p_l + N_5$	$R_5\ p_l + Q_5$
SH	N_2	Q_2

Once the data distribution and the number of processors at each level are determined, communication structures to be inserted between levels can be selected. For each level, feasible communication structures, that are valid for all arrays at that level, are identified considering the references to the same arrays in the upper and lower levels of the code. Then, a common feasible communication structure is selected. In this work, instead of selecting the communication structure randomly, the one that produces the minimum transfer time is selected.

In order to minimize the program execution time, the best number of processors for each level of the code and the best alignment for the arrays that are referred to at each level of the code must be determined. A parallel code might consist of many levels. Furthermore, for each level there might be different possibilities for alignment, degree of parallelism and communication structure. For this reason, the problem of finding the optimal parallel code configuration becomes highly complex.

III. Genetic Algorithms For Parallel Code Optimization

Genetic Algorithms have been used for solving many difficult problems [19] that were introduced by J. Holland in 1975 [20]. Given a parallel code, consisting of L levels and a distributed memory parallel machine with P processors, parallel code optimization (PCO) can be stated as finding the best mapping of P processors and D alignments of data decomposition patterns at each level, reducing the total expected execution time of the algorithm. The search space ranges up to $(PD)^L$, assuming the best communication structure for the level. Although not considered in here, the size of a level and selecting the levels to combine may also be used as other optimization parameters.

Machine parameters are obtained by Benchmarking as explained in [16] and fed into the GA Solver for execution time optimization as shown in Figure 5. The *Alignment Parser* parses the given sequential code to be parallelized, in a C like language. It determines all possible array alignments in the code and generates the related data decomposition patterns at all levels, handing them over to the GA solver. Finally, GA solver produces the best assignment of alignments and number of processors at each level.

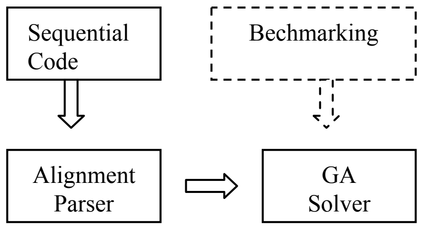


Figure 5. Flowchart showing how parallel code optimization problem is solved by GA

A. GA Components

Each individual is a list of alleles, where each locus represents a level in the given code. Alleles consist of two parts: number of processors to be used at a level and the data alignment. Figure 6 demonstrates an example individual, having 5 levels.

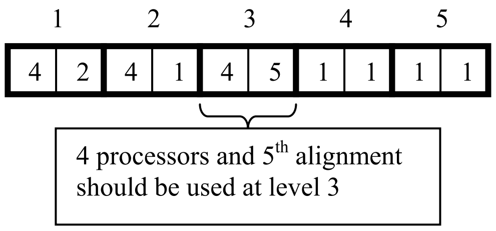


Figure 6. An example individual representing a candidate solution for a PCO problem with 5 levels

Two types of Genetic Algorithms are implemented as a solver: steady state genetic algorithms (SSGA) and transgenerational genetic algorithms (TGGA). Population size is chosen to be proportional to the length of an individual. Fitness function indicates the total execution time as shown in Equation 4.

The best communication structure is chosen at each level among all possibilities. This step can be considered as a hill climbing step. This process can be applied, since the contribution of the communication structure to the total execution time at each level is independent.

SSGA and TGGA both utilize linear ranking strategy for selecting parents and elitist replacement strategies. In SSGA, two worst individuals in the population are deleted and both offspring are injected in their places. In TGGA, the best of the offspring combined with the previous population forms the next generation.

SSGA visits two new states at each generation, while the number of states that TGGA visits is two times the individual length, determining the number of evaluations.

Different crossover operators are tested: traditional one point crossover (1PTX), two-point crossover (2PTX) and uniform crossover (UX). Traditional mutation is used, randomly perturbing an allele, assigning a random value to the number of processors and the decomposition pattern.

Runs are terminated, whenever the expected fitness is reached or the maximum number of generations is exceeded.

IV. Experimental Data

Experiments are performed using three data sets produced from two different algorithms.

A. Hessenberg Reduction

In the first and second data sets, the parallel algorithm in [21] for reducing matrices to Hessenberg form is used. One iteration of the Hessenberg reduction is represented as follows:

$$A = (I - V Z V^T)^T A (I - V Z V^T),$$

where A, Z, V are NxN matrices. Its parallel implementation has 5 levels as shown in Figure 7.

Levels 1, 2, 4 and 5 perform matrix multiplication operation where the execution time increases with N^3, and level 3 consists of a subtraction operation where the execution time increases with N^2.

```
for (i=0; i<N; ++i)  /* Level 1 */
   for (j=0; j<N; ++j) {
      a[i][j]=0;
      for (k=0; k<N; ++k)

a[i][j]=a[i][j]+V[i][k]*Z[k][j];
         /* VZ */
   }
 for (i=0; i<N; ++i)  /* Level 2 */
   for (j=0; j<N; ++j) {
      b[i][j]=0;
      for (k=0; k<N; ++k)

b[i][j]=b[i][j]+a[i][k]*V[j][k];
             /* VZV^T */
   }
 for (i=0; i<N; ++i)  /* Level 3 */
   for (j=0; j<N; ++j)
        c[i][j]=I[i][j]-b[i][j];
        /* I- VZV^T */
 for (i=0; i<N; ++i) /* Level 4 */
   for (j=0; j<N; ++j) {
      d[i][j]=0;
      for (k=0; k<N; ++k)

d[i][j]=d[i][j]+c[k][i]*A[k][j];
          /* (I- VZV^T)^T A */
   }
 for (i=0; i<N; ++i) /* Level 5 */
   for (j=0; j<N; ++j){
      e[i][j]=0;
      for (k=0; k<N; ++k)
        e[i][j]=e[i][j]+d[i][k]*c[k][j];
        /* (I- VZV^T)^T A (I- VZV^T) */
   }
```

Figure 7. Implementation of Hessenberg reduction

Hessenberg reduction is chosen due to its simple nature. At each level there are three possible alignments and three choices for the number of processors (1, 4 and 16). In order to be able to test GA using a known result, all 9^5 possible combinations of alignments and the number of processors are computed to determine the optimum configuration.

When the problem size N is chosen as 1024, the problem is communication intensive; therefore the best fitness is achieved when the algorithm is run on one processor regardless of the decomposition pattern.

However when N is chosen as 10240, the problem becomes computation intensive and the best fitness is observed utilizing 16 processors at each level, as marked in Figure 8. Note that there are two optimal configurations having the best fitness.

Above data sets are used during the initial experiments, denoted as H1 and H2, with N=1024 and N=10240, respectively. Considering the size of the search space and the number of optimal points having the same fitness value in the search space, H1 is a simpler problem than H2 to solve.

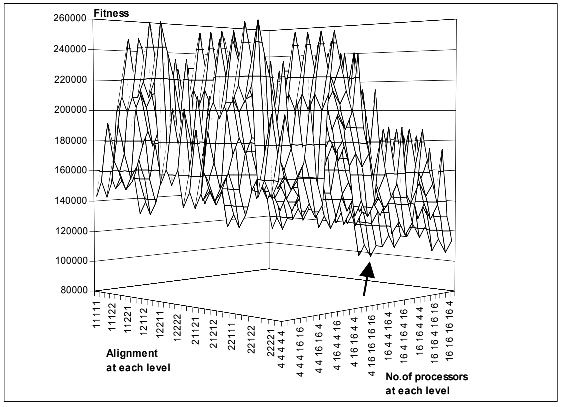


Figure 8. Part of the fitness landscape for H2, N=10240, considering {4, 16} as possible number of processors and {1, 2} as possible alignment ids, where arrow points out the best fitness.

B. Dongarra's Benchmark

The third data set consists of 11 loops selected from the test loops prepared by J. Dongarra for parallelizing compilers, found at site:

http://www.netlib.org/benchmark/parallel

This problem involves 12 matrices of sizes n_1 and n_2, and vectors of size n_2. Matrix addition, matrix multiplication, vector-matrix multiplication, assignment, etc. operations are executed on different matrices and vectors (Table III).

In this case, for all levels, fitness increases with n_1xn_2, except for level 1 where fitness increases with n_1^3. When n_1 and n_2 are chosen as 10240, level 1 becomes computation intensive, and therefore it can be estimated that it will have the best time for 16 processors. Level 2 also has the same arrays as level 1, therefore to minimize the communication time, 16 processors may be feasible. Operations in the other levels are not computation intensive and they have a different set of arrays than levels 1 and 2. Therefore, it is expected that 1 processor may be the best choice for these levels.

TABLE III. DETAILS FOR SELECTED LOOPS FROM DONGARRA'S BENCHMARK SET.

l	Arrays	Array Operations
1	X2, Y2, Z2	Matrix Multiplication
2	X2, Y2, Z2	Assignment, 2D-Array Multiplication
3	A2, B2, C2	2D-Array Addition
4	A2, B2, C2, S	2D-Array Addition, Scalar Multiplication
5	A2, B2, C2, D1	1D and 2D-Array Multiplication and Addition
6	A2, B2, C2, D2	2D-Array Addition
7	A2, B2, C2, D2, A1	1D and 2D-Array Multiplication and Addition
8	A2, B2, C2, D2, E2, F2, A1, B1	1D and 2D-Array Multiplication and Addition
9	A2, B2, C2, D2, E2, F2	2D-Array Addition and Subtraction
10	A2, B2, D2, A1	1D and 2D-Array Multiplication and Addition
11	A2, B2, D2, A1	1D and 2D-Array Multiplication and Addition

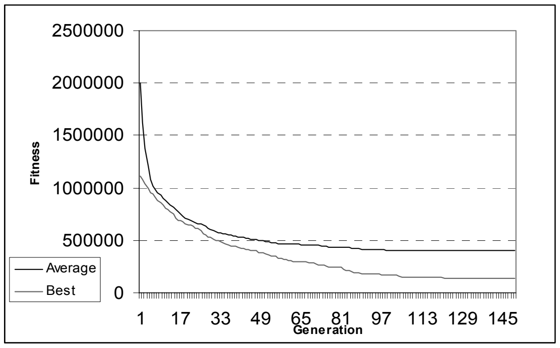
V. EXPERIMENTAL RESULTS

Runs are performed on an Intel Pentium 4, 1.7GHz machine. Each experiment is repeated for 100 times. *Success rate*, μ, is the ratio of the number of runs returning the optimal solution to the total number of runs. Initial experiments are performed to test crossover operators and different types of GAs using Hessenberg reduction, yielding the experimental results, summarized in Table IV. Runs are terminated for H1 and H2, as explained in Section III. A. Since the expected fitness is not known for D1, a run is terminated whenever the maximum number of generations is exceeded.

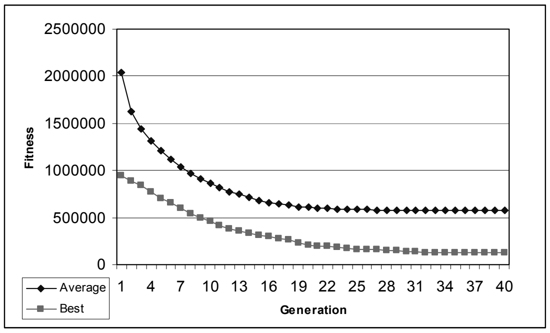
TABLE IV. TEST RESULTS FOR H1 AND H2 USING DIFFERENT GA TYPES AND CROSSOVER (XOVER) OPERATORS, WHERE μ DENOTES SUCCESS RATE, ρ DENOTES AVERAGE NUMBER OF GENERATIONS PER RUN, σ DENOTES STANDARD DEVIATION.

Data	**GA Type**	**Xover**	**μ**	**(ρ**	**σ)**
H1	SSGA	1PTX	1.00	68.40	94.20
H1	SSGA	2PTX	1.00	69.49	97.31
H1	SSGA	UX	1.00	65.23	91.51
H1	TGGA	1PTX	1.00	17.13	18.51
H1	TGGA	2PTX	1.00	17.44	18.87
H1	TGGA	UX	1.00	17.44	18.96
H2	SSGA	1PTX	0.81	4908.99	7865.88
H2	SSGA	2PTX	0.84	4737.81	7287.54
H2	SSGA	UX	0.82	4694.87	7470.72
H2	TGGA	1PTX	0.79	898.02	1506.45
H2	TGGA	2PTX	0.81	816,31	1460,61
H2	TGGA	UX	0.80	956.62	1527.07

GA solves H1 much faster than H2, as expected. The solutions are found for H1 and H2 in tens of generations and in seconds as shown in Figure 8.



(a)



(b)

Figure 9. Plot of average and best fitness at each generation averaged over 100 runs for solving H1 using (a) SSGA and (b) TGGA with 1PTX.

While solving H2, GA sometimes gets stuck at the same local optimum, arranging expected number of processors and related alignments at the first two levels and failing at the rest. Although, these instances are not that many, the individuals tend to become alike. Parallel code optimization problem can be formulized as a multiobjective problem. Premature convergence is a common issue, while applying Genetic Algorithms on some multiobjective problems, as in [22]. As the size of the search space of a PCO problem instances increases and the number of global optima in the search space decreases, we expect that this issue will arise and become a burden.

Comparing the number of evaluations, SSGA performs slightly better than TGGA. For example, SSGA with 1PTX requires 136 (68.40*2) evaluations on average, while TGGA with 1PTX requires 171 (17.13*10) for solving H1. Furthermore, crossover operators establish approximately the same performance. For this reason, D1 is tested using only SSGA with 1PTX.

Results supported our predictions about the solution, as explained in Section IV. B. 50% of the runs yield the same solution for D1, emphasizing the problem is computation intensive for the first two levels. Figure 10 shows this final solution generated by SSGA with 1PTX. Therefore, the first two levels are mapped to 16 processors and the same decomposition patterns are used in order to minimize the communication. Other levels are computed on one processor. As the computation is sequential on one processor, communication patterns do not have any effect.

```
[  1] 16, 2     [  7]  1, -
[  2] 16, 2     [  8]  1, -
[  3]  1, -     [  9]  1, -
[  4]  1, -     [ 10]  1, -
[  5]  1, -     [ 11]  1, -
[  6]  1, -
```

Figure 10. A solution of D1 obtained by SSGA with 1PTX, where each entry is in the following form: `[level] number of processors, alignment id`.

VI. Conclusions And Future Work

Our experiments show that SSGA is more promising than TGGA for solving the parallel code optimization problem. SSGA with 1PTX successfully generated a solution for Dongarra's benchmarks (D1).

Speed is an important issue for this problem. If it takes too long time to get optimal settings for a code, users may prefer utilizing their intuition for parallelizing their codes. Note that success rates for the simpler instances of the problem are much better than the success rate for D1. Furthermore, the initial indication of premature convergence is received. Hence, a diversification scheme can be added to improve the GA solver. There is a variety of approaches, such as crowding [23], or hyper mutation [24] for keeping the population diversified. Different hill climbing techniques can also be developed as a part of the GA solver.

References

[1] J. Li, M. Chen, "Index Domain Alignment: Minimizing Costs of Cross-referencing between Distributed Arrays", *Third Symp. on the Frontiers of Massively Parallel Computation*, pp. 424-433, 1990.

[2] J. Anderson, M. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines", *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 112-125, 1993.

[3] R. Bixby, K. Kennedy, U. Kremer, "Automatic Data Layout using 0-1 Integer Programming", *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques* (PACT'94), 1994.

[4] K. Knobe, J. Lucas, G. Steele, "Data Optimizations: Allocation of Arrays to Reduce Communication on SIMD machines", *Journal of Parallel and Distributed Computing*, no. 8, 102-118, 1990.

[5] D. Palermo, *Compiler Techniques for Optimizing Communication and Data Distribution for Distributed Memory Multicomputers*, PhD Thesis, University of Illinois at Urbana-Champaign, 1996.

[6] J. Ramanujan, P. Sadayappan, "Compile-time Techniques for Data Distribution in Distributed Memory Machines", *IEEE Tr. on Parallel and Distributed System*s, pp. 472-482, 1991.

[7] T. Rauber, G. Runger, "Deriving Array Distributions by Optimization Techniques", *J. Supercomputing*, vol. 15, pp. 271-293, 2000.

[8] S. Wholey, "Automatic Data Mapping for Distributed Memory Parallel Computers", *Proc. of Int. Conf. on Supercomputing*, 1992.

[9] M. Wolfe, *High Performance Compilers for Parallel Computers, Addison-Wesley*, 1996.

[10] M.E. Wolf, M.S. Lam, "A Data Locality Optimizing Algorithm", *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 30-44, 1991.

[11] K. Ikudome, G. Fox, A. Kolawa, J. Flower, "An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers", *Proc. of 5th Distributed Memory Computing Conference*, pp. 1105-1114, 1990.

[12] E. Onbasioglu, L. Ozdamar, "Optimization of Data Distribution and Processor Allocation Problem using Simulated Annealing", *The Journal of Supercomputing*, vol.25, pp. 237-253, 2003.

[13] T. Chen, J. Sheu, "Communication-Free Data Allocation Techniques for Parallelizing Compilers on Multicomputers", *IEEE Tr. on Parallel and Distributed Systems*, pp. 924-938, 1994.

[14] C.H. Huang, P. Sadayappan, "Communication-Free Hyperplane Partitioning of Nested Loops", *J. Parallel and Distributed Computing*, vol. 19, pp. 90-102, 1993.

[15] M. Mace, *Memory Storage Patterns in Parallel Processing*, Kluwer Academic, 1987.

[16] E. Onbasioglu, Y. Paker, "A Comparative Workload-based Methodology for Performance Evaluation of Parallel Computers", *Future Generation Computer Systems*, vol. 12, pp. 512-545, 1997.

[17] J. Mohan, *Performance of Parallel Programs*, PhD dissertation, Dept. of Computer Science, Carnegie-Mellon University, 1984.

[18] P.Lee, "Efficient algorithms for Data Distribution on Distributed Memory Parallel Computers", *IEEE Tr. on Parallel and Distributed Systems*, vol. 8, no. 8, pp. 825-839, 1997.

[19] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading (MA), 1989.

[20] J. H. Holland, *Adaptation in Natural and Artificial Systems*, Univ. Mich. Press, 1975.

[21] J. Choi, J.J. Dongarra, D.W. Walker, "The design of a parallel, Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal and Bidiagonal Form", *Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing*, pp. 98-111, 1994.

[22] A. Alkan, E. Ozcan, "Memetic Algorithms for Timetabling", *Proc. of IEEE Congress on Evolutionary Computation*, pp. 1796-1802, 2003.

[23] K.A. De Jong., *An Analysis of the Behavior of a Class of GeneticAdaptive Systems*, PhD Thesis, University of Michigan, Ann Arbour, MI, 1975.

[24] H.G. Cobb, "An investigation into the use of hypermutation as an adaptive operator in Genetic Algorithms Having Continuous, Time-dependent Nonstationary Environment", NRL Memorandum Report 6760, 1990.