# A PARALLEL GENETIC ALGORITHM CODE FOR SCHEDULING PROBLEMS

Bryan A. Norman
and
James C. Bean
Dept of Industrial and Operations Engineering
The University of Michigan
1205 Beal Avenue
Ann Arbor, Michigan 48109-2117

Technical Report 94-23

September 1994
Revised October 1994

# A Genetic Algorithm Code for Scheduling Problems: Parallel Computing Version *

Bryan A. Norman
James C. Bean

Department of Industrial and Operations Engineering
University of Michigan, Ann Arbor, MI 48109-2117

Version 1.2

October 17. 1994

*This work was supported in part by the National Science Foundation under Grant DDM-9018515 and DDM-9308432 to the University of Michigan.

# Contents

# 1 Introduction

This report is a documentation and a user manual for the MPL code, **genjssp_par**, developed for finding solutions to job shop scheduling problems (JSSP). The code can be applied to JSSPs with the following complexity:

1. Non-zero ready times.
2. Due dates.
3. Job shop or open shop structure.
4. Multiple, non-identical machines.
5. Routing flexibility for jobs.
6. Sequence dependent setup times.
7. Tooling constraints. Different jobs may compete for the same tool and/or the changeover from one tool to another may induce sequence dependent setup times.
8. The objective function is a combination of regular measures.

The code presented here does not consider precedence constraints among the jobs. However, it can be modified in the manner described in Norman[1994] to handle precedence constraints. It is also assumed that setups only occur when there is a tooling change. The code could be readily modified to accommodate other types of setups by making changes to the **chromosome_evaluation** function. The objective function considered in the program is the basic tardiness measure. Weighted tardiness and more complicated objective function measures can be considered by simply modifying the **chromosome_evaluation** function. It is assumed that the problem has a minimization objective. If the objective is maximization the following changes should be made: change the **chromosome_evaluation** function to reflect the correct objective function, sort each generation's objective function values in descending order rather than ascending order,

modify the code sections that collect information about the best solution found and the first solution found within 5% of the lower bound.

The code implements a genetic algorithm (GA) to search the problem space. The GA utilizes the random keys encoding described in Bean[1994] and a coarse-grained parallel population structure. For this particular implementation, alleles are represented by integers where the 3 least significant digits contain job number information and the fourth least significant digit contains machine assignment information. Computational tests show that **genjssp_par** finds good solutions, within 5% of provable lower bounds, to 300 job problems containing the previously listed complexities. See Norman and Bean[1994] and Norman[1994] for more information.

# 2 User Manual

## 2.1 Code Location

The source code and other related files are placed on the anonymous FTP machine called *freebie.engin.umich.edu* under the directory

**/pub/misc/ga_sched**

The files have been compressed and stored in one file, named **genjssp.zip**, using the Unix zip utility. To retrieve these files, the user should use the ftp program on a machine with tcp/ip connectivity to the Internet. One should connect to *freebie* with the ftp command, and log in with account name *anonymous* and his/her electronic mail address as the password.

In the following ftp session, the file **genjssp.zip** is retrieved.

```
dexter% ftp freebie.engin.umich.edu
Connected to knob2.engin.umich.edu.
220 knob2.engin.umich.edu FTP server (Version 5.60) ready.
Name (freebie.engin.umich.edu:user_id): anonymous
```

```
331 Guest login ok, send ident as password.
Password:  <<< Your Email Address Here >>>
230 Guest login ok, access restrictions apply.
ftp> cd pub/misc/ga_sched
250 CWD command successful.
ftp> get genjssp.zip
200 PORT command successful.
150 Opening ASCII mode data connection for genjssp.zip (29708 bytes).
226 Transfer complete.
local: genjssp.zip remote: genjssp.zip
127515 bytes sent in 1 seconds (1.2e+02 Kbytes/s)
ftp> quit
221 Goodbye.
dexter%
```

Once the file **genjssp.zip** is retreived the file should be unzipped. This will place several files in the current directory including: example problem data and the source code, example parameter file, sample output, and documentation for both the serial and parallel versions of the genetic algorithm. There is also a file titled **FILE.INFO** that contains a brief description of each of the files.

## 2.2 Compiling and Running

This parallel code was written to run on a MasPar MP-1 Massively Parallel Computer. The machine used in this research has 1024 processors (referred to as processor elements or PE's). This computer has a SIMD architecture and the variables that reside on each PE are referred to as plural variables. The main difference between this code and serial codes is that operations are performed on each chromosome simultaneously. Users interested in more specific information about the MasPar hardware and MPL programming language should refer to MasPar Documentation[1991]. To produce an executable program on a

MasPar machine using the optimizer option (Omax), type

**mpl genjssp_par.m -Omax -o genjssp_par**

The executable **genjssp_par** is invoked by the command

**genjssp_par**

There are two input files for the program **genjssp_par**. The first input file, named **genjssp_data**, contains the job data. The format for this file is as follows. The first line contains the values of the total number of jobs, $N$, the total number of machines, $M$, and the total number of tools, $T$. The next $N$ lines contain specific data for each job. For each job $i = 1, \ldots N$, this data includes: job number, $i$, tool requirement, $t_i$, ready time, $r_i$, due time, $d_i$, processing time, $p_i$, number of machines that can process the job, $nm_i$, and the list of machine numbers that can process the job, $m_{i,j}$ for $j = 1, \ldots nm_i$. The remaining $T$ lines contain the setup times, $s_{k,l}$, for switching from tool type $k$ to tool type $l$ for $k, l = 1, \ldots T$. The input file format is given below

$$
\begin{array}{cccccccc}
N & M & T & & & & & \\
1 & t_1 & r_1 & d_1 & p_1 & nm_1 & m_{1,1}, \ldots m_{1,nm_1} \\
2 & t_2 & r_2 & d_2 & p_2 & nm_2 & m_{2,1}, \ldots m_{2,nm_2} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\
N & t_N & r_N & d_N & p_N & nm_N & m_{N,1}, \ldots m_{N,nm_N} \\
s_{1,1} & s_{1,2} & \cdots & s_{1,T} & & & & \\
s_{2,1} & s_{2,2} & \cdots & s_{2,T} & & & & \\
\vdots & \vdots & \vdots & \vdots & & & & \\
s_{T,1} & s_{T,2} & \cdots & s_{T,T} & & & &
\end{array}
$$

The second input file, **genjssp_parameters**, contains problem specific parameter settings for the GA. These parameters are discussed in more detail in Section 2.3. The order of the parameters in the file **genjssp_parameters** should match the order found in the discussion in Section 2.3.

## 2.3 Setting Parameters

The maximum problem and population sizes are given. They are used for dimensioning arrays.

| | |
|---|---|
| **Max_jobs** | The maximum number of jobs. |
| **Max_machines** | The maximum number of machines. |
| **Max_pop** | The maximum population size. |
| **Max_tools** | The maximum number of tools. |

They are specified at the beginning of the code with **#define** statements (see section 3.1). If these limits are violated, **genjssp_par** terminates with a descriptive error message. In this case, the user should increase the limit(s) and recompile the code.

The following are more problem-specific parameters, which are read from the input file **genjssp_parameters**. A brief description of each parameter is provided below, for more detailed information see Norman[1994].

| | |
|---|---|
| **total_pop_size** | The total population size. Empirically 1 to 4 times the number of jobs works well. For small problems use a minimum size of 100. Larger population sizes produce better results but require longer running times to converge to a solution. |
| **max_generations** | The maximum number of generations that the GA will run. Longer runs produce better solutions but require longer running times. |
| **clone_frac** | The elitest strategy is implemented by copying the **number_clones** best solution from each subpopulation into the next generation. Subpopulation size multiplied by **clone_frac** yields the value for **number_clones**. Empirically a value of .04 - .05 for **clone_frac** works well. |
| **crossover_prob** | The crossover probability used in the Bernoulli crossover operator. A value of 0.7 typically performs well. |

**immig_frac** The fraction of each subpopulation that are immigrants. Empirically values in the range .02 - .03 work well.

**immig_type_rate** The fraction of the immigrants that are not strongly biased. Empirically a value of 0.6 works well.

**ready_weight** The biasing weight assigned to ready time. Based on empirical results, the weights for the ready time and the due time should be approximately equal. Empirically a value of 1.0 works well.

**due_weight** The biasing weight assigned to due time.

**strong_bias_frac** The fraction of each subpopulation that will be strongly biased initially. Empirically a value of .7 works well.

**num_subpops** The number of subpopulations. Empirically 4 works well. If the **total_pop_size** is very large (>1024) this value could be increased. There is a trade-off between the benefit of having more subpopulations and the liability of the subpopulations becoming too small to search effectively.

**commun_frac** The fraction of each subpopulation to communicate between subpopulations. Empirically values in the range .03 - .04 work well. In general, the value should be less than the value of **clone_frac** and enough less that **number_clones - number_to_commun** > 0.

**gen_to_commun** The number of generations between occurrences of communication between subpopulations. Empirically a value of 20 works well.

**startseed** The initial random number seed.

**endseed** The ending random number seed. The GA will run (endseed-startseed)/2000 times. The 2000 is introduced because each PE requires a random number seed and there are 1024 PE's on the MasPar machine used in this research (2000 was used instead of 1024 to keep the value a round number).

**lb_value** The value of a lower bound (assuming the objective is minimization) for the problem.

## 2.4 An Example

Consider the following example with 10 jobs, 2 machines, and 3 tools. The job data is provided below.

$$N = 10 \quad M = 2 \quad T = 3$$

| $i$ | $t_i$ | $r_i$ | $d_i$ | $p_i$ | $nm_i$ | $m_{i,1}$ | $m_{i,2}$ |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 4.68 | 2.00 | 1.12 | 2 | 1 | 2 |
| 2 | 1 | 1.25 | 2.50 | 0.61 | 2 | 1 | 2 |
| 3 | 2 | 1.50 | 2.28 | 1.91 | 2 | 1 | 2 |
| 4 | 2 | 1.75 | 4.12 | 0.43 | 2 | 1 | 2 |
| 5 | 2 | 1.99 | 5.13 | 0.77 | 2 | 1 | 2 |
| 6 | 3 | 2.24 | 3.89 | 1.05 | 2 | 1 | 2 |
| 7 | 3 | 2.49 | 4.27 | 0.49 | 2 | 1 | 2 |
| 8 | 3 | 2.74 | 6.74 | 2.31 | 2 | 1 | 2 |
| 9 | 2 | 3.12 | 5.54 | 1.10 | 2 | 1 | 2 |
| 10 | 1 | 2.78 | 4.11 | 0.41 | 2 | 1 | 2 |

$$s_{1,1} = 0.00 \quad s_{1,2} = 0.68 \quad s_{1,3} = 1.42$$
$$s_{2,1} = 0.75 \quad s_{2,2} = 0.00 \quad s_{2,3} = 0.99$$
$$s_{3,1} = 1.81 \quad s_{3,2} = 1.12 \quad s_{3,3} = 0.00$$

The corresponding input file for the job data, here called **genjssp_data**, is set up for this problem as follows:

```
10   2   3
 1   1   4.68   2.00   1.12   2   1   2
 2   1   1.25   2.50   0.61   2   1   2
 3   2   1.50   2.28   1.91   2   1   2
 4   2   1.75   4.12   0.43   2   1   2
 5   2   1.99   5.13   0.77   2   1   2
 6   3   2.24   3.89   1.05   2   1   2
 7   3   2.49   4.27   0.49   2   1   2
```

```
   8   3   2.74   6.74  2.31  2  1  2
   9   2   3.12   5.54  1.10  2  1  2
  10   1   2.78   4.11  0.41  2  1  2
0.00 0.68 1.42
0.75 0.00 0.99
1.81 1.12 0.00
```

Note that the format of the data file may be such that each data element is in a separate line as long the elements are in the right order. An example input file for the GA parameters, **genjssp_parameters**, is as follows:

```
100 250 .06 .70 .04 .6 1.0 0.9 .7 2 .04 20 1 20000 11.36
```

Here is a sample session and the corresponding output. For this example the variable **print_freq** was set equal to 5 and only the results for the first random seed are shown. For this random seed, the optimum value of 11.37 was found.

## Example Session.

```
auch% genjssp_par

The number of jobs is 10.
The total number of machines is 2.
The total number of tools is 3.

-----------------GA-PARAMETER VALUES----------------------

total_pop_size =        100
max_generations =       250
clone_frac =          0.060
crossover_prob =       0.70
immig_frac =          0.040
immig_type_rate =      0.60
ready_weight =         1.00
due_weight =           0.90
strong_bias_frac =     0.70
num_subpops =             2
commun_frac =         0.04
gen_to_commun =         20
startseed =              1
```

endseed = 20000
lb_value = 11.36

----------GENERATION RESULTS----------

| Generation Number | Best Solution Found |
|---|---|
| 5 | 14.36 |
| 10 | 13.54 |
| 15 | 13.54 |
| 20 | 13.54 |
| 25 | 13.54 |
| 30 | 12.98 |
| 35 | 12.98 |
| 40 | 12.98 |
| 45 | 12.98 |
| 50 | 11.43 |
| 55 | 11.43 |
| 60 | 11.37 |
| 65 | 11.37 |
| 70 | 11.37 |
| 75 | 11.37 |

----------SUMMARY DATA----------

The total Dpu time = 5.05 seconds.

The best solution found is 11.37

The best sol. of 11.37 found in gen 60 required 4.04 seconds.

Lb+5 percent solution 11.43 found in gen 48 required 3.24 seconds.

----------FINAL SOLUTON----------

| Job Number | Machine Assignment | Completion Time | Tardiness For This Job | Total Tardiness |
|---|---|---|---|---|
| 3 | 1 | 3.41 | 1.13 | 1.13 |
| 2 | 2 | 1.86 | 0.00 | 1.13 |
| 4 | 1 | 3.84 | 0.00 | 1.13 |
| 7 | 2 | 3.77 | 0.00 | 1.13 |
| 5 | 1 | 4.61 | 0.00 | 1.13 |

| | | | | |
|---|---|---|---|---|
| 9 | 1 | 5.71 | 0.17 | 1.30 |
| 10 | 1 | 6.87 | 2.76 | 4.06 |
| 6 | 2 | 4.82 | 0.93 | 4.99 |
| 1 | 1 | 7.99 | 5.99 | 10.98 |
| 8 | 2 | 7.13 | 0.39 | 11.37 |

# 3 Code Documentation

## 3.1 Basic Definitions and Data Structure

```
#include <mpl.h>
#include <stdio.h>
#include <math.h>
#include <mp_libc.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

#define Max_jobs 500            /* Max number of jobs.     */
#define Max_machines 20         /* Max number of machines. */
#define Max_pop 1025            /* Max population size.    */
#define Max_tools 50            /* Maximum number of tools.*/

#define maxint 2147483648       /* Max integer value.      */

/* General GA variables. */
int total_pop_size;             /* Total population size.  */
int max_generations;            /* Maximum number of gens  */
int number_clones;              /* No. clones in a subpop. */
float clone_frac;               /* Clone frac. for each subpop.   */
int number_immig;               /* No. immigrants in a subpop.    */
float immig_frac;               /* Immig. frac. for each subpop.  */
float crossover_prob;           /* Crossover probability.  */
int number_jobs;                /* Total number of jobs.   */
int number_tools;               /* Total number of tools.  */
int total_number_machines;      /* Total number of machs.  */
int startseed,endseed;          /* Random seed values.     */
int min_target = 0;             /* Minimum target value.   */
float lb_value;                 /* Lower bound value.      */
float immig_type_rate;          /* Prob. for a type of immigrant.*/
plural int rank1;               /* Rank within each subpop.*/
plural int rank2;               /* Rank within total pop.  */
double data_to_rank[Max_pop+1]; /* Array which is ranked.  */

/* General GA variables */
plural int chrom_alleles[3][Max_jobs+1]; /* Chromosome alleles.           */
plural double chrom_fitness[3];          /* Actual chrom. fitness.        */
plural double pseudo_chrom_fitness[3];   /* Pseudo-fitness for subpop rank.*/
```

```
/* Subpopulation variables. */
int num_subpops;                /* Number of subpops.          */
int subpop_size;                /* Size of each subpop.        */
plural int subpop_number;       /* Subpop no. for each chrom. */
float commun_frac;              /* Frac. of subpop to commun. */
int num_to_commun;              /* No. of chrom. to commun.    */
int gen_to_commun;              /* No. of gens. until commun. */

/* Plural variables that store job data            */
/* in each PE in order to improve the run time.    */
plural float due[Max_jobs+1];
plural proctime[Max_jobs+1];
plural float ready[Max_jobs+1];
plural int tool[Max_jobs+1];

/* Variables for the chromosome_evaluation() and                          */
/* print_final_solution() functions.                                      */
plural int   temp_alleles[Max_jobs+1];    /* Temp. array to sort RKs.     */
plural float machendtime[Max_machines+1]; /* When machine next available.*/
plural float toolavail[Max_tools+1];      /* When tool next available.    */
plural int   toollastmach[Max_tools+1];   /* Where a tool last used.      */
plural float setupvalue[Max_tools+1][Max_tools+1]; /* Setup times.        */
float setup[Max_tools+1][Max_tools+1];    /* Setup times.                 */

/* Biasing variables */
float ready_weight;             /* Ready time bias weight.         */
float due_weight;               /* Due time bias weight.           */
float raw_bias=0;               /* Raw bias.                       */
double adj_bias=0;              /* Adjusted bias.                  */
float true_bias[Max_jobs+1];    /* Array of adjusted biases.       */
float strong_bias_frac;         /* Frac. of each subpop strongly biased */
int strong_bias;                /* No. per subpop strongly biased.*/

/* Other variables. */
int job[Max_jobs+2];            /* Indices for machine array.      */
int machine[2000];              /* Alternative machines for a job.*/
int best_solution_gen=0;        /* Gen. that best solution found. */
int lb_solution_gen=0;          /* Gen. that Lb+5% solution found.*/
float best_solution;            /* Best solution found.            */
float lb_solution;              /* Lb+5% solution found.           */
float lblatetem;                /* Temp. var. to find lb_solution.*/
```

```
float total_time;              /* Total program time.            */
float total_time_to_lb;        /* Time until Lb+5% sol. found.   */
float total_time_to_best;      /* Time until best solution found.*/
int best[Max_pop+1];           /* Holds PE no. for subpop. rank. */

FILE *fpl;   /* Opens the data file. */

/* Defining the structures for the data file. */

struct facilities
{
int tool;
int index;
int num_machines;
float ready;
float due;
float proctime;
};

struct facilities data[Max_jobs];
```

A given generation consists of **total_pop_size** solutions (or individuals) split into **num_subpops** subpopulations. Each individual has information stored in three different arrays. The array **chrom_alleles** contains the allele values for the individual. The actual fitness is contained in **chrom_fitness** and the pseudo fitness (used to rank within the different subpopulations) is contained in **pseudo_chrom_fitness**.

## 3.2 Input Function

The function **readdata** reads the problem data and GA parameters and assigns the appropriate values to singular variables.

```
void readdata()

/*********************************************************/
/* The function readdata() reads in the problem data     */
/* from the file genjssp_data                            */
/* and then creates two vectors that relate machine      */
/* compatibility with each job.                          */
/*********************************************************/
{
int i=1,j,k=0;

/* Open the job data file.                 */
fpl= fopen("genjssp_data", "r");

/* If the data file does not exist, print a message.     */
if (fpl == NULL)
   {
   printf("Unable to open genjssp_data \n");
   return;
   }

/* Read in data file size variables and check that they   */
/* do not exceed the defined limits.                      */
fscanf(fpl,"%d %d %d",&number_jobs,&total_number_machines,&number_tools);
printf("genjssp\_par \n\n");
printf("The number of jobs is %d.\n",number_jobs);
printf("The total number of machines is %d.\n",total_number_machines);
printf("The total number of tools is %d.\n\n",number_tools);

if(number_jobs>Max_jobs)
   {
   printf("Error: Job Limit Exceeded.\n");
   printf("Data = %d and Limit = %d.\n",number_jobs,Max_jobs);
   exit(1);
   }
if(total_number_machines>Max_machines)
   {
```

```
   printf("Error: Machine Limit Exceeded.\n");
   printf("Data = %d and Limit = %d.\n",total_number_machines,Max_machines);
   exit(1);
   }
if(number_tools>Max_tools)
   {
   printf("Error: Tool Limit Exceeded.\n");
   printf("Data = %d and Limit = %d.\n",number_tools,Max_tools);
   exit(1);
   }

/* The job data information is read in as a structure.   */
/* As the data is read in, two single-column arrays,   */
/* job[] and machine[] that relate machine      */
/* compatibility with jobs are created.              */
job[0]=0;
while (i<=number_jobs)
   {
   fscanf(fpl,"%d%d%f%f%f%d",&data[i].index,&data[i].tool,
      &data[i].ready,&data[i].due,&data[i].proctime,&data[i].num_machines);
   for(j=1;j<=data[i].num_machines;j++)
      {
      fscanf(fpl,"%d",&machine[k]);
      k++;
      }
   job[i]=k;
   i++;
   } /* end of while statement */
for(j=1;j<=number_tools;j++)
   for(k=1;k<=number_tools;k++)
      fscanf(fpl,"%f",&setup[j][k]);
fclose(fpl);

/* Open the GA parameters file.            */
fpl= fopen("genjssp_parameters", "r");

/* If the data file does not exist, print a message.    */
if (fpl == NULL)
   {
   printf("Unable to open genjssp_parameters \n");
   return;
   }
```

```
/* Read in the GA parameter data.          */
fscanf(fpl,"%d%d%f%f%f%f%f%f%f%d%f%d%d%d%f",&total_pop_size,
   &max_generations,&clone_frac,&crossover_prob,&immig_frac,
   &immig_type_rate,&ready_weight,&due_weight,&strong_bias_frac,&num_subpops,
   &commun_frac,&gen_to_commun,&startseed, &endseed,&lb_value);

printf("-----------------GA-PARAMETER VALUES----------------------\n\n");
printf("total_pop_size = %10d \n",total_pop_size);
printf("max_generations = %9d \n",max_generations);
printf("clone_frac = %14.3f \n",clone_frac);
printf("crossover_prob = %10.2f \n",crossover_prob);
printf("immig_frac = %14.3f \n",immig_frac);
printf("immig_type_rate =  %8.2f \n",immig_type_rate);
printf("ready_weight = %12.2f \n",ready_weight);
printf("due_weight = %14.2f \n",due_weight);
printf("strong_bias_frac = %8.2f \n",strong_bias_frac);
printf("num_subpops = %13d \n",num_subpops);
printf("commun_frac = %13.2f \n",commun_frac);
printf("gen_to_commun = %11d \n",gen_to_commun);
printf("startseed = %15d \n",startseed);
printf("endseed = %17d \n",endseed);
printf("lb_value = %16.2f \n\n",lb_value);

/* Check that the population size does not exceed its limit. */
if(total_pop_size>Max_pop)
   {
   printf("Error: Population Size Limit Exceeded.\n");
   printf("Data = %d and Limit = %d.\n",
      total_pop_size,Max_pop);
   exit(1);
   }

/* Convert percentages to integer values. The .01 is to
   catch round-off errors.  */
subpop_sizetotal_pop_siznum_subpops;
number_clones= clone_frac*subpop_size+.01;
number_immig = immig_frac*subpop_size+.01;
strong_bias = strong_bias_frac*subpop_size+.01;
num_to_commun= commun_frac*subpop_size+.01;
}
```

The function **convertdata** copies singular data that is needed on each PE to plural variables.

```
void convertdata()

/*********************************************************/
/* The function convertdata() converts the singular      */
/* job data to a plural copy to reside on each PE.       */
/*********************************************************/
{
int i,j,k;

for(i=1;i<=number_jobs;i++)
   {
   ready[i]=data[i].ready;
   due[i]=data[i].due;
   proctime[i]=data[i].proctime;
   tool[i]=data[i].tool;
   }
for(j=1;j<=number_tools;j++)
   for(k=1;k<=number_tools;k++)
      setupvalue[j][k]=setup[j][k];
}
```

## 3.3 Genetic Algorithm Functions

There is one main GA function called **genetic** and four other functions referred to as **initialize_population**, **reproduction**, **communication** and **chromosome_evaluation**. **Genetic** first calls **initialize_population** which initializes some parameters and randomly generates an initial population of solutions. Then, it calls iteratively **reproduction** which, given an initial generation, reproduces a new one using the process of elitist reproduction, Bernoulli crossover, and the immigration operator described in Norman and Bean[1994]. **Communication** is called every **gen_to_commun** generations to share chromosomes between the different subpopulations. The function **chromosome_evaluation** evaluates a solution given its index in the current population. The function **chromosome_evaluation** is called by **initialize_population** and **reproduction**.

```
void chromosome_evaluation(m)

/*********************************************************/
/* The function chromosome_evaluation(m) evaluates       */
/* the fitness of a chromosome.                          */
/*********************************************************/
int m;
{
plural int job_to_sched;
plural int oldtooltype[Max_machines+1],newtype[Max_machines+1];
plural int i,mach,temp;
plural float setuptime;

/* Initialize.                                           */
chrom_fitness[m]=0;
for(i=0;i<=total_number_machines;i++) machendtime[i] = 0;
for(i=0;i<=total_number_machines;i++) oldtooltype[i] = 0;
for(i=1;i<=number_tools;i++)
   {
   toolavail[i]=0;
   toollastmach[i]=0;
   }

/* Sort the random key values for the jobs.              */
for (i=1;i<= number_jobs;i++)
    temp_alleles[i] = chrom_alleles[m][i]+i;
heapsort(number_jobs);

/* Construct a semi-active schedule based on the sorted */
/* order of the random keys.            */
for (i=1;i<=number_jobs;i++)
   {
/* Determine the job number and machine assignment.              */
/* Note that the job number information                          */
/* is contained in the 3 least significant digits of the allele  */
/* (this needs to be increased if the job number exceeds 1000)   */
/* and the machine assignment is contained in the fourth least   */
/* significant digit of the allele (this could be extended if the */
/* machine number exceeds 10).                                   */
   temp=temp_alleles[i]%10000;
   mach=(temp)/1000;
   job_to_sched = temp-(mach)*1000;
```

```
/* Calculate the appropriate setup time.      */
  if(tool[job_to_sched]==oldtooltype[mach] &&
      toollastmach[tool[job_to_sched]]==mach)
     setuptime=0;
  else
     setuptime=setupvalue[oldtooltype[mach]][tool[job_to_sched]];

/* Calculate the starting time and subsequent    */
/* completion time for each job.          */
   if (machendtime[mach]<toolavail[tool[job_to_sched]])
       machendtime[mach]=toolavail[tool[job_to_sched]];
   machendtime[mach]+=setuptime;
   if (machendtime[mach]<ready[job_to_sched])
       machendtime[mach]=ready[job_to_sched];
   machendtime[mach] = toolavail[tool[job_to_sched]] =
       machendtime[mach] + proctime[job_to_sched];

/* Update the tool indicators.          */
   toollastmach[tool[job_to_sched]]=mach;
   oldtooltype[mach]=tool[job_to_sched];

/* Determine the tardiness.            */
   if (machendtime[mach] - due[job_to_sched]>0)
      chrom_fitness[m] += machendtime[mach]-due[job_to_sched];
   }
}
```

The function **initialize_population** determines subpopulation data, calculates the biasing values for each job, and creates an initial generation of chromosomes.

```
void initialize_population()

/*********************************************************/
/* The function initialize_population() initializes     */
/* the chromosomes, evaluates their fitness and         */
/* performs the necessary ranking.                      */
/* It also determines the bias values from              */
/* their ready and due weights.                         */
/*********************************************************/
{
int j;
```

```
/* Indicate which subpop a given processor is in. */

subpop_size=total_pop_size/num_subpops;
subpop_number=iproc/subpop_size;

/* Calculate the biasing values for each job.      */
for (j=1;j<=number_jobs;j++)
   {
   raw_bias=(ready_weight*proc[1].ready[j]+720)+(due_weight*proc[1].due[j]);
   raw_bias /= 240.0;
   true_bias[j] = pow(10.0,raw_bias);
   }

/* Create the initial population of chromosomes.   */
for (j=1;j<=number_jobs;j++)
   {
   chrom_alleles[0][j] = triag(0.0,true_bias[j]);
   if(iproc-subpop_size*subpop_number<strong_bias)
      chrom_alleles[0][j] = (plural int)(0.5*(true_bias[j]));
   chrom_alleles[0][j] *= 10000;
   chrom_alleles[0][j] += (machine[job[j-1]+
                (pick(job[j]-job[j-1])-1)])*1000;

/* A check to see if the bias exceeds the feasible range */
/* for the variable type or any other data problem       */
/* that would lead to a negative allele value.           */

   if(chrom_alleles[0][j]<0)
      {
      printf("raw_bias= %f true_bias[%d]= %f\n",
       raw_bias,j,true_bias[j]);
      }
   }
chromosome_evaluation(0);

/* Rank the current population. */
rank2=rankf(chrom_fitness[0]);

/* Second, rank the subpopulations. */
pseudo_chrom_fitness[0]=1000000.0*subpop_number+chrom_fitness[0];
rank1 = rankf(pseudo_chrom_fitness[0]);
```

```
best[rank1]=iproc;
}
```

The function **reproduction** first employs the elitest strategy by leaving the **number_clones** best solutions in each subpopulation unchanged. Second, the Bernoulli crossover operation is performed. The solutions are ranked within each subpopulation and the **number_immigrants** worst solutions in each subpopulation are replaced using the immigration operator.

```
void reproduction()

/*************************************************************/
/*  The function reproduction() performs the generational    */
/*  changes for the GA. Both crossover and immigration are   */
/*  performed within this function.                          */
/*************************************************************/
{
int j;
plural int crossover_partner1,crossover_partner2;
plural float immigration_variate;
plural int tempchrom_alleles;
plural float n;

/* Select crossover partners. Notice that the number_clones */
/* pe's with the best objective function measure are        */
/* cloned into the next generation (an elitest strategy).   */

if(iproc-subpop_number*subpop_size<number_clones)
   crossover_partner1=crossover_partner2=best[iproc];
else
   {
   crossover_partner1=pick(subpop_size)-1;
   crossover_partner2=pick(subpop_size)-1;
   crossover_partner1+=subpop_number*subpop_size;
   crossover_partner2+=subpop_number*subpop_size;
   crossover_partner1=best[crossover_partner1];
   crossover_partner2=best[crossover_partner2];
   }

/* Fetch the crossover partner's gene values and copy    */
```

```
/* them to the PE. The data must be stored in array      */
/* positions 1 and 2 because other pe's are copying the  */
/* data from array position 0. After all the data has    */
/* been copied the data is moved from position 2 to      */
/* position 0.                                           */

ss_rfetch(crossover_partner1,&chrom_alleles[0][1],&chrom_alleles[1][1],
  number_jobs*sizeof(int));
ss_rfetch(crossover_partner2,&chrom_alleles[0][1],&chrom_alleles[2][1],
  number_jobs*sizeof(int));

for(j=1;j<=number_jobs;j++)
   chrom_alleles[0][j]=chrom_alleles[2][j];

/* Perform Bernoulli crossover on all the pe's that are      */
/* supposed to be performing it (all the pe's that           */
/* do not contain clones) with probability crossover_prob.   */

if (iproc-subpop_number*subpop_size>=number_clones)
   for(j=1;j<=number_jobs;j++)
      {
      n = urand();
      if (n>=crossover_prob) ;
      else
         {
         tempchrom_alleles=chrom_alleles[0][j];
         chrom_alleles[0][j]=chrom_alleles[1][j];
         chrom_alleles[1][j]=tempchrom_alleles;
         }
      }

/* Evaluate the solutions on each PE and retain the     */
/* best one.                                            */
chromosome_evaluation(0);
chromosome_evaluation(1);
if(iproc-subpop_number*subpop_size>=number_clones)
   {
    if (chrom_fitness[1] < chrom_fitness[0])
      {
      for (j=1;j<=number_jobs;j++)
          chrom_alleles[0][j] = chrom_alleles[1][j];
      chrom_fitness[0] = chrom_fitness[1];
```

```
      }
   }

/* Rank the current population. */
rank2=rankf(chrom_fitness[0]);

/* Second, rank the subpopulations. */
pseudo_chrom_fitness[0]=1000000.0*subpop_number+chrom_fitness[0];
rank1 = rankf(pseudo_chrom_fitness[0]);
best[rank1]=iproc;

/* Perform the immigration operation. */

if (rank1-subpop_number*subpop_size>subpop_size-number_immig-1)
   {

/* Determine whether the variate will be strongly  */
/* biased or not.                    */
   immigration_variate=urand();
   if(immigration_variate<immig_type_rate)
/*    Create a non-strongly biased immigrant.     */
      for (j=1;j<=number_jobs;j++)
          {
          chrom_alleles[0][j] = triag(0,true_bias[j]);
          chrom_alleles[0][j]*=10000;
          chrom_alleles[0][j] += (machine[job[j-1]+
                  (pick(job[j]-job[j-1])-1)])*1000;
          }
   else
/*    Create a strongly biased immigrant.     */
      for (j=1;j<=number_jobs;j++)
          {
          chrom_alleles[0][j]=((plural int)(0.5*true_bias[j]))*10000;
          chrom_alleles[0][j] += (machine[job[j-1]+
                  (pick(job[j]-job[j-1])-1)])*1000;
          }
   }
}
```

The function **communicate** shares chromosomes between the different subpopulations after a fixed number of generations. Note that the sharing always occurs in the

same direction between the same two subpopulations.

```
void communicate()

/*************************************************************/
/* The function communicate() shares the best solutions      */
/* from different subpopulations. The best                   */
/* num_to_commun solutions                                   */
/* from the receiving subpopulation are replaced by the best */
/* num_to_commun solutions from the                          */
/* sending subpopulation.                                    */
/*************************************************************/
{
plural int commun_partner,m;

/* Each subpopulation i (except the first one) replaces    */
/* its best num_to_commun chromosomes with the best        */
/* num_to_commun chromosomes from population i-1.          */
/* Subpopulation 1 replaces its best                       */
/* num_to_commun chromosomes with the best                 */
/* num_to_commun chromosomes from population num_subpops-1.*/
if(iproc-subpop_number*subpop_size < num_to_commun)
   {
   if(iproc<subpop_size)
      commun_partner=iproc+((num_subpops-1)*subpop_size);
   else
      commun_partner=iproc-subpop_size;
   ss_rfetch(commun_partner,&chrom_alleles[0][1],&chrom_alleles[2][1],
        number_jobs*sizeof(int));
   for(m=1;m<=number_jobs;m++)
               chrom_alleles[0][m]=chrom_alleles[2][m];
   ss_rfetch(commun_partner,&chrom_fitness[0],&chrom_fitness[2],
        sizeof(double));
   chrom_fitness[0]=chrom_fitness[2];
   pseudo_chrom_fitness[0]=1000000.0*subpop_number+chrom_fitness[0];
   }

rank1=rankf(pseudo_chrom_fitness[0]);
best[rank1]=iproc;
}
```

The function **genetic** runs while the stopping critera are not met (i.e., **stop**=0). There are three stopping criteria: the GA has run for **max_generations** number of generations, the best solution is equal to a lower bound, and the best solution has remained unchanged for 15 generations and the **number_of_clones** best solutions within the entire population have the same objective function value.

```
void genetic()

/*********************************************************/
/* The function genetic() runs the GA until a stopping  */
/* criterion is met.                                     */
/*********************************************************/
{
int generation_count,stop,print_freq = 1,dputimer_reset=80;

/* Initialize. */
stop = 0;
generation_count = 0;
printf("-----------------GENERATION RESULTS---------------------\n\n");
printf("Generation Number  Best Solution Found \n");

/* Continue with new generations until a stopping */
/* criterion is met.                              */
while (1-stop) {
   generation_count++;

/* Communicate between subpopulations when necessary.   */
   if(generation_count%gen_to_commun==0)
      communicate();

/* Reset timer when necessary.   */
   if(generation_count==dputimer_reset*(generation_count/dputimer_reset))
      {
      total_time+=dpuTimerElapsed();
      dpuTimerStart();
      }

/* Perform reproduction to create a new generation. Only use */
/* the PE's that are necessary.   */
   if(iproc<total_pop_size)
      reproduction();
```

```
 /* If it is time, print the current best solution.   */
   if(generation_count==print_freq*(generation_count/print_freq))
      if(rank2==0)
         {
         printf("%10d",generation_count);
         p_printf("              %.2f \n",chrom_fitness[0]);
         }

/* Check to see if the best solution and lower + 5%   */
/* solution need to be updated.                       */
   if(rank2==0)
      {
      if(chrom_fitness[0]+.01<best_solution)
         {
         best_solution=reduceAddf(chrom_fitness[0]);
         best_solution_gen=generation_count;
         total_time_to_best=total_time+dpuTimerElapsed();
         }
      if(chrom_fitness[0]+.01 < lblatetem*1.05)
         {
         lb_solution=reduceAddf(chrom_fitness[0]);
         lb_solution_gen=generation_count;
/*       Reset lblate so only find the first sol. within 5%. */
         lblatetem=0.0;
         total_time_to_lb=total_time+dpuTimerElapsed();
         }
       }

/* Check the stopping criteria.                */
   if (rank2==0) if (chrom_fitness[0]<=min_target) stop=1;
   if(generation_count >= max_generations) stop = 1;
   if(best_solution_gen+15<=generation_count)
      if(generation_count>50)
         if(rank2==number_clones-1)
            if(chrom_fitness[0]-.01<best_solution)
               stop = 1;
   }
}
```

## 3.4 Main Function and Utilities

The function **print_final_solution** prints the final solution to the screen. The output contains the machine assignment, completion time and tardiness for each job.

```
void print_final_solution()

/*********************************************************/
/* The function print_final_solution() prints out the   */
/* final GA solution.                                    */
/*********************************************************/
int m;
{
plural int job_to_sched;
plural int oldtooltype[Max_machines],newtype[Max_machines];
plural int i,mach,temp;
plural float setuptime;
plural double tardiness;

/* Initialize.                                  */
chrom_fitness[m]=0;
for(i=0;i<=total_number_machines;i++) machendtime[i] = 0;
for(i=0;i<=total_number_machines;i++) oldtooltype[i] = 0;
for(i=1;i<=Max_tools;i++)
   {
   toolavail[i]=0;
   toollastmach[i]=0;
   }
printf("--------------------FINAL SOLUTION-------------------------\n\n");
printf(" Job     Machine    Completion  Tardiness For     Total  \n");
printf(" Number  Assignment     Time       This Job      Tardiness\n");

/* Sort the random key values for the jobs.             */
for (i=1;i<= number_jobs;i++)
    temp_alleles[i] = chrom_alleles[m][i]+i;
heapsort(number_jobs);

/* Construct a semi-active schedule based on the sorted */
/* order of the random keys.            */
for (i=1;i<=number_jobs;i++)
   {
/* Determine the job number and machine assignment.              */
```

```
/* Note that the job number information                              */
/* is contained in the 3 least significant digits of the allele      */
/* (this needs to be increased if the job number exceeds 1000)       */
/* and the machine assignment is contained in the fourth least       */
/* significant digit of the allele (this could be extended if the    */
/* machine number exceeds 10).                                       */
   temp=temp_alleles[i]%10000;
   mach=(temp)/1000;
   job_to_sched = temp-(mach)*1000;

/* Calculate the appropriate setup time.      */
  if(tool[job_to_sched]==oldtooltype[mach] &&
      toollastmach[tool[job_to_sched]]==mach)
     setuptime=0;
  else
     setuptime=setupvalue[oldtooltype[mach]][tool[job_to_sched]];

/* Calculate the starting time and subsequent    */
/* completion time for each job.          */
   if (machendtime[mach]<toolavail[tool[job_to_sched]])
       machendtime[mach]=toolavail[tool[job_to_sched]];
   machendtime[mach]+=setuptime;
   if (machendtime[mach]<ready[job_to_sched])
       machendtime[mach]=ready[job_to_sched];
   machendtime[mach] = toolavail[tool[job_to_sched]] =
       machendtime[mach] + proctime[job_to_sched];

/* Update the tool indicators.          */
   toollastmach[tool[job_to_sched]]=mach;
   oldtooltype[mach]=tool[job_to_sched];

/* Determine the tardiness.  */

   tardiness=0;
   if (machendtime[mach] - due[job_to_sched]>0)
      chrom_fitness[m]+=tardiness=machendtime[mach]-due[job_to_sched];

/* Print out the info for each job. */

   p_printf(" %4d        %2d        %6.2f       %6.2f        %6.2f \n",
      job_to_sched,mach,machendtime[mach],tardiness,chrom_fitness[m]);
   }
```

```
}
```

The function **urand** randomly generates uniform $(0,1)$ variates using the function **p_random**.

```
plural float urand()

/*********************************************************/
/* The function urand() generates a                      */
/* uniform (0,1) variate.                                */
/*********************************************************/
{
plural float p;

p = p_random();
p = p/maxint;
return p;
}
```

The function **pick** randomly generates integer numbers using the function **p_random**.

```
plural int pick(n)

/*************************************************************/
/*  The function pick(n) returns an integer in the range     */
/*  1 to n.                                                  */
/*************************************************************/
int n;
{
plural int p1;
plural float p2;

p2 = p_random();
p1 = p2*n/maxint;
p1=p1+1;
if (p1<1) p1 = 1;
if (p1>n) p1 = n;
return p1;
}
```

The function **triag** randomly generates an integer that is triagonally distributed between lo_value and hi_value with mode equal to their average.

```
plural int traig(lo_value,hi_value)

/*************************************************************/
/*  The function triag(lo_value,hi_value) returns an         */
/*  integer that is triangularly distributed in the range    */
/*  [lo_value, hi_value].                                    */
/*************************************************************/
float lo_value;
float hi_value;
{
plural float tempgene1;
plural int tempgene2;
float midpoint;

midpoint=0.5*(hi_value+lo_value);
tempgene1=urand();
if(tempgene1<(float)(midpoint-lo_value)/(float)(hi_value-lo_value))
   tempgene2=lo_value+p_sqrt(midpoint-lo_value)*
             p_sqrt((hi_value-lo_value)*tempgene1);
else
   tempgene2=hi_value-p_sqrt(hi_value-midpoint)*
             p_sqrt((hi_value-lo_value)*(1.0-tempgene1));
return tempgene2;
}
```

The function **heapsort** sorts the array temp_alleles in ascending order. It is called by **chromosome_evaluation** in order to sort the random key values.

```
void heapsort(num_to_sort)

/*************************************************************/
/*  Heapsorts the array temp_alleles[n] in increasing order. */
/*  A heapsort is used because it is supposed to be the      */
/*  most effective on the MIMD architecture for lists        */
/*  with a size of a few hundred items.                      */
/*************************************************************/
int num_to_sort;
{
plural int stop;
plural int l,ir,j,i;
plural int rra;
```

```
l=(num_to_sort >> 1) +1;
ir = num_to_sort;

for(;;)
   {
   if(l>1)
      rra=temp_alleles[--l];
   else
      {
      rra=temp_alleles[ir];
      temp_alleles[ir]=temp_alleles[1];
      if( --ir == 1)
         {
         temp_alleles[1]=rra;
         return;
         }
      }
   i=l;
   j=l << 1;
   while (j<=ir)
      {
      if (j < ir && temp_alleles[j] < temp_alleles[j+1])
         ++j;
      if (rra < temp_alleles[j])
         {
         temp_alleles[i]=temp_alleles[j];
         j += (i=j);
         }
      else j=ir+1;
      }
   temp_alleles[i]=rra;
   }
}
```

The internal MasPar function **rankf** sorts plural data in ascending order. It is used to sort the chromosomes based on their fitness. It is used for sorting within subpopulations (rank1) and for sorting the entire population (rank2). In order to sort within the subpopulations, fitness is converted to pseudo-fitness by multiplying the subpopulation number of each member of a subpopulation by a large constant (1000000.0 in this code) and adding that to the actual fitness value. A sort based on pseudo-fitness orders the

chromosomes first by subpopulation and then by fitness within each subpopulation. Note that for different problems a larger constant may be required depending on the range of objective function values.

The **main** function calls **readdata**, **genetic**, prints final solution information and calls **print_final_solution**.

```
main()

/*********************************************************/
/* Beginning of main program.                            */
/*                                                       */
/*********************************************************/
{
int i,j;
int loopseed;
plural unsigned int seed;

readdata();
convertdata();

/* Run the GA for a number of times based on the values */
/* of startseed and endseed.                            */
if(iproc<total_pop_size)
for(loopseed=startseed;loopseed<=endseed;loopseed+=2000)
   {
   seed=loopseed;
   for(i=0;i<=nproc;i++)
       proc[i].seed+=proc[i].iproc;
   p_srandom(seed);

/* Reset the timing values and the best and lower bound */
/* solution values.                                     */
   dpuTimerStart();
   total_time=0;
   total_time_to_lb=0;
   total_time_to_best=0;
   best_solution=100000; lb_solution=100000;
   best_solution_gen=0; lb_solution_gen=0;
   lblatetem=lb_value;

   initialize_population();
```

```
   genetic();

/* Calculate and print out the total time that the GA   */
/* ran, the value of the best solution and the time     */
/* required to find it, the time required to get within */
/* 5% of the lower bound, and the final best solution.  */
printf("\n--------------------SUMMARY DATA-------------------------\n\n");
   printf("\nThe total Dpu time = %.2f seconds.\n",total_time+dpuTimerElapsed());
   rank2=rankf(chrom_fitness[0]);
   if(rank2==0)
      {
      p_printf("\nThe best solution found is %.2f\n",chrom_fitness[0]);
      printf("\nThe best sol. of %.2f found in gen %d ",
               best_solution,best_solution_gen);
      printf("required %.2f seconds.\n ",total_time_to_best);
      printf("\nLb+5 percent solution %.2f found in gen %d ",
                lb_solution,lb_solution_gen);
      printf("required %.2f seconds.\n ",total_time_to_lb);
      printf("\n");

      print_final_solution(0);
      printf("\n");
      }
    printf("\n");
    }/*end seed loop*/
}
```

## REFERENCES

Bean, J. C. [1994], "Genetic Algorithms and Random Keys for Sequencing and Optimization," **ORSA Journal on Computing,** Vol. 6, Spring 1994, 154-160.

MPL User Guide [1991]. Document Part Number 9302-0101, Revision A1, MasPar Computer Corporation, Sunnyvale, CA.

Norman, B. A. and J. C. Bean [1994]. "Random Keys Genetic Algorithm for Job Shop Scheduling," Technical Report 94-5, Department of Industrial and Operations Engineering, University of Michigan.

Norman, B. A. [1994]. Forthcoming Ph.D. Dissertation, Department of Industrial and Operations Engineering, University of Michigan.