THE THIRD INTERNATIONAL WORKSHOP ON PARALLEL ARCHITECTURES AND BIOINSPIRED ALGORITHMS

THE NINETEENTH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES

EDITED BY
J. MANUEL COLMENAR
DANIEL LOMBRAÑA GONZÁLEZ

ARCHITECTURE AND TECHNOLOGY OF COMPUTING SYSTEMS GROUP,
UNIVERSIDAD COMPLUTENSE DE MADRID

UNIVERSITY OF EXTREMADURA

UNIVERSIDAD COMPLUTENSE DE MADRID

ISBN 978-84-693-6141-2

THE THIRD INTERNATIONAL WORKSHOP ON PARALLEL ARCHITECTURES AND BIOINSPIRED ALGORITHMS

SEPTEMBER 11-15 2010

VIENNA (AUSTRIA)

EDITED BY

J. MANUEL COLMENAR

DANIEL LOMBRAÑA GONZÁLEZ

PRINTED IN UNIVERSIDAD COMPLUTENSE DE MADRID, SPAIN

SEPTEMBER 2010

ADDITIONAL COPIES OF THE PROCEEDINGS OF THE WPABA ARE AVAILABLE FROM:
JOSÉ L. RISCO MARTÍN
DEPARTMENT OF COMPUTER ARCHITECTURE AND AUTOMATION
UNIVERSIDAD COMPLUTENSE DE MADRID
C/ PROF. JOSÉ GARCÍA SANTESMASES, S/N.
28040 MADRID (SPAIN)

THE THIRD INTERNATIONAL WORKSHOP ON PARALLEL ARCHITECTURES AND BIOINSPIRED ALGORITHMS

VIENNA (AUSTRIA)

SEPTEMBER 11-15 2010

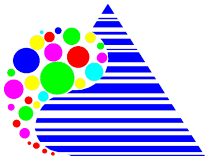
ORGANIZED BY

ARCHITECTURE AND TECHNOLOGY OF COMPUTING SYSTEMS GROUP, UNIVERSIDAD COMPLUTENSE DE MADRID

UNIVERSITY OF EXTREMADURA

UNIVERSIDAD COMPLUTENSE DE MADRID

SPONSORED BY



PACT 2010 - THE NINETEENTH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES

EDITORS

J. MANUEL COLMENAR
C. E. S. FELIPE II (ARANJUEZ CAMPUS)
UNIVERSIDAD COMPLUTENSE DE MADRID
C/ CAPITÁN, 39
28300 ARANJUEZ (SPAIN)
jmcolmenar@cesfelipesegundo.com

DANIEL LOMBRAÑA GONZÁLEZ
DEPARTMENT OF COMPUTER TECHNOLOGY AND COMMUNICATIONS
UNIVERSITY OF EXTREMADURA
CENTRO UNIVERSITARIO DE MÉRIDA
C/STA. TERESA DE JORNET, 38
06800 MÉRIDA (SPAIN)
daniellg@unex.es

THE NINETEENTH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, PACT 2010

GENERAL CHAIR

VALENTINA SALAPURA (IBM T.J. WATSON RESEARCH CENTER)

PROGRAM CHAIRS

MICHAEL GSCHWIND (IBM SYSTEMS & TECHNOLOGY GROUP)

JENS KNOOP (VIENNA UNIVERSITY OF TECHNOLOGY)

THE THIRD INTERNATIONAL WORKSHOP ON PARALLEL ARCHITECTURES AND BIOINSPIRED ALGORITHMS, WPABA 2010

GENERAL CO-CHAIRS

JOSÉ L. RISCO-MARTÍN

FRANCISCO FERNÁNDEZ

JUAN LANCHARES

WPABA 2010 International Program Committee

David Atienza, EPFL (Switzerland)
Erick Cantú-Paz, Yahoo Inc. (USA)
J. Manuel Colmenar, ArTeCS Research Group (Spain)
Francisco Fernández, University of Extremadura (Spain)
Oscar Garnica, Universidad Complutense de Madrid (Spain)
Steven Gustafson, General Electric Global Research Center (USA)
J. Ignacio Hidalgo, Universidad Complutense de Madrid (Spain)
Juan Lanchares, Universidad Complutense de Madrid (Spain)
Daniel Lombraña González, University of Extremadura (Spain)
Nouredine Melab, INRIA (France)
Sanaz Mostaghim, University of Karlsruhe (Germany)
José L. Risco-Martín, Universidad Complutense de Madrid (Spain)
Giandomenico Spezzano, ICAR-CNR (Italy)

GENERAL CO-CHAIRS' MESSAGE

Welcome to the Third International Workshop on Parallel Architectures and Bioinspired Algorithms, WPABA 2010

Dear Participants, Dear Guests,

It is our great pleasure to welcome you to the third Workshop on Parallel Architectures and Bioinspired Algorithms (WPABA'10) held in conjunction with PACT 2010. This year the conference takes place in Vienna (Austria), at the historic Austrian Academy of Sciences. This historic building is located in the city center and is within walking distance of many of the historic places, such as Stephansplatz, the central point in Vienna, the historic city synagogue, the Ring Boulevard and the Imperial Palace.
Vienna is one of Europe's most historic cultural centers. At the crossroads of east and west, north and south, Vienna was the residence of the German Kings and Holy Roman Emperors, and the capital of the Austrian-Hungarian Empire. The scenic Danube Valley is home to historic castles, monasteries and Roman settlements and border posts dating back over 2000 years.

The aim of this Workshop is to join a larger number of researchers interested in the synergies arising from different but related fields: Parallel Computer Architectures, Parallel and Distributed Computing, and Bioinspired Algorithms. The interaction among researchers within these fields is becoming usual. On the one hand, parallel architecture designers, and in general, system designers are starting to bear in mind Bioinspired Algorithms as general optimization tools. These algorithms comprise a set of heuristics that can help to optimize a wide range of tasks required for Parallel and Distributed Architectures to work efficiently: balancing computer load, fault-tolerance and dependability, thermal-aware design, NoC design, and other related problems. On the other hand, the application of Bioinspired Algorithm to solve real-world problems has shown that they need high computation power. Parallel Architectures and Distributed Systems have offered an interesting alternative to sequential counterparts since improvements on parallel architectures are allowing running computing intensive Bioinspired Algorithms for solving many difficult engineering problems.

We hope you enjoy the Workshop, and have a nice stay in Vienna.

WPABA 2010 General Co-Chairs
José L. Risco-Martín
Francisco Fernández
Juan Lanchares

ACKNOWLEDGEMENTS

The WPABA 2010 International Program Committee selected the papers for the Conference among all submissions and we expected a very successful event based on their efforts; so we would like to thank all the authors as well as the IPCs and reviewers for their review process.
A special thank to the organizations, institutions and societies that are supporting and technically sponsoring the event: Universidad Complutense de Madrid, University of Extremadura, Architecture and Technology of Computing Systems Group and The International Conference on Parallel Architectures and Compilation Techniques. Finally, we would like to thank all the Workshop Organization Supporters, specially the technical staff who prepared the web for the workshop: Adrián Bravo, Ezequiel Lara and Luis Canet.

LOCAL ORGANIZATION COMMITTEE

Index

Effective Mutation Operator for Nurse Scheduling by Cooperative GA and Its Parallel Processing

Makoto Ohki
Division of Information and Electronics, Graduate School of Tottori University
101, 4 Koyama-Minami, Tottori, Tottori 680-8552 Japan
+81 857 31 5231

mohki@ele.tottori-u.ac.jp

ABSTRACT

This paper proposes effective operators for Cooperative Genetic Algorithm (CGA) to be applied to a real nurse scheduling problem. The nurse scheduling is very complex task, because many requirements must be considered for the scheduling. In fact, the nurse schedule is still made by the hand of a manager, or a chief nurse, in many general hospitals because the schedule generated by machine cannot be satisfied. It is hard to revise the schedule too. In our investigation, even a veteran manager spends one or two weeks for nurse scheduling. On the other hand, CGA is very powerful tool to optimize such a combinatorial problem. We apply CGA to the nurse scheduling problem in this paper. In our algorithm, the number of nurses at each shift must be satisfied as a strong constraint. Other constraints are defined as weak constraints or penalties of the population. CGA is superior in ability for local search, but is often stagnated as the unfavorable situation because it is inferior to ability for global search. To improve these problems, we propose several effective mutation operators for the nurse scheduling. The mutation yields small changes in the population when the optimization of the schedule stagnates. Then the population is able to escape from a local minimum area. In the real hospital, the change of the schedule occurs frequently. Such the change of the shift schedule yields various problems, for example, imbalance of the number of the holidays and the number of the attendance. Such problems fall the nursing level of the whole nurse organization. Therefore, such an imbalance by the change of the nurse schedule must be broken off. This paper describes a technique to re-optimize the nurse schedule that the shift schedule has been changed. In this case a more powerful operator is necessary. We propose a Multi-Branched Mutation (MBM) operator. By using the MBM operator, concurrency of the optimization process can be naturally extracted. Therefore, we have implemented parallel processing of the nurse scheduling.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Sequencing and scheduling, H.4.1 [**Office Automation**]: Time management (e.g., calendars, schedules), I.1.2 [**Algorithms**]: Nonalgebraic algorithms, I.2.8 [**Problem Solving, Control Methods, and Search**]: Scheduling, J.3 [**LIFE AND MEDICAL SCIENCES**]: Medical information systems

General Terms

Algorithms, Management

Keywords

Nurse Scheduling, Cooperative Genetic Algorithm, Mutation Operator, Multi-Branched Mutation

1. INTRODUCTION

General hospitals consist of several sections such as the internal medicine department and the pediatrics department, etc. About fifteen to thirty nurses work in each section. A section manager arranges a shift schedule of all the nurses in her/his section every month. The manager considers more than fifteen requirements for arranging the shift schedule. Such the arrangement of the schedule, or the nurse scheduling, is very complex task. In our investigation, even a veteran manager spends one or two weeks for the nurse scheduling. This means a great loss of work force and time. Therefore, computer software for the nurse scheduling has recently come to be required in the general hospitals [1-5]. In fact, the nurse schedule is still made by the hand of the manager in many general hospitals because the schedule generated by machine cannot be satisfied. It is hard to revise the schedule too.

In this paper, we discuss about generation and optimization of the nurse schedule by using the Cooperative Genetic Algorithm (CGA). The conventional CGA optimizes the nurse schedule by using only crossover operator, because the crossover has been considered as the only one method which keeps consistency of relation between chromosomes in the CGA. We have already proposed an effective mutation operator keeping such consistency [6-9] for the CGA. This mutation operator effectively works for the optimization. This mutation operator is activated depending on the optimization speed. However, there are many parameters to define this mutation operator. To improve this difficulty, we propose a simple mutation operator activated periodically.

In the real case, some nurses come to their office on a day different from the original schedule because of circumstances of other nurse or an emergency. We discuss such a case that the nurse schedule has been changed in the past weeks of the current month [6,7]. By such a change of the shift schedule, various

 ISBN 978-84-693-6141-2

inconveniences occur, for example, imbalance of the number of the holidays and the number of the attendance. Such an inconvenience causes the fall of the nursing level of the whole nurse organization. Therefore, such the inconvenience should be broken off. Considering the change of the shift schedule whenever one week passes, the shift schedule is re-optimized in remaining weeks of the current month. We discuss a technique to improve such an inconvenience by re-optimizing the schedule and a multiplex divergence type mutation operator called as a Multi-Branched Mutation (MBM) operator, as a good operator of the search efficiency. This new operator is suitable for the parallel processing of the nurse scheduling. By performing the optimization of the shift schedule in a manner of parallel computation, better solution is acquired in shorter time. We have implemented such the parallel computation of the nurse scheduling by using MPI technology.

By using the MBM operator, concurrency of the optimization process can be naturally extracted. Therefore, we have implemented parallel processing of the nurse scheduling. This parallel processing technique is executed by using MPI™ technology. In the night time, there are many PCs which are inactive in the hospital. The parallel optimization is executed by using these inactive PCs.

2. NURSE SCHEDULING

2.1 Genetic Coding of Nurse Schedule

An individual and a population in the CGA for the nurse scheduling are defined as shown in Fig.1. The individual consists of the sequence of the duty symbols. The duty sequence consists of thirty fields, since one month includes thirty days in the practical example. Each individual expresses one-month schedule of the nurse *i*. There are not two or more individuals including the same nurse's schedule in the population. In the CGA, the population denotes the whole schedule.

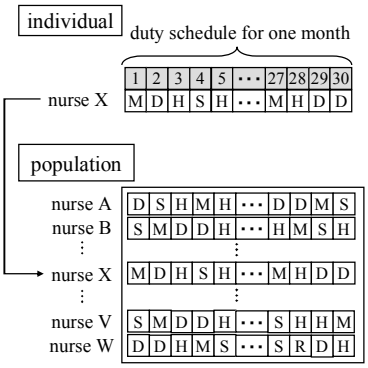


Figure 1. An individual coded in chromosome giving shift schedule of the nurse, X, for one month and a population including one-month schedules of all nurses. D, S, M and H denotes a day time duty, a semi-night duty, midnight duty and holiday respectively.

2.2 Performing the Nurse Schedule

For arranging the nurse schedule, the clinical director must consider many requirements. For example, meeting, training, requested holiday, these must be accepted, where we assume that all the requested holidays have been confirmed by the director. Semi-night and midnight duty should be fairly arranged to all nurses. And it is prohibited to make nurses work for more than six consecutive days. We have summarized all the requirements into the twelve penalties. These penalties are classified into four penalty groups.

We define a penalty function on shift pattern as the following equation,

$$H_1 = \sum_{i=1}^{M} \left(h_{11} F_{1i} + h_{12} F_{2i} + h_{13} F_{3i} \right) \tag{1}$$

where F_{1i}, F_{2i} and F_{3i} denote the following penalty functions about the shift pattern.

We classify consecutive duty patterns for three days into four categories as shown in Table 1. In this table, Meeting and training are handled as day time duty, and requested holiday is handled as holiday. The first category denotes a top priority pattern, and its penalty value is defined to zero. The second category denotes a priority pattern, and its penalty value is defined to one. The third category denotes a compromised pattern, and its value is defined to two. The final category denotes a prohibited pattern, and its penalty value is defined to five. Comparing whole shift schedule of the nurse *i* with Table 1, the penalty is given by the following equation.

$$F_{1i} = \sum_{j=1}^{D-1} p_{ij} + p_{iD} \tag{2}$$

where p_{ij} denotes the penalty value as given by Table 1 and p_{iD} denotes the penalty value defined to the final two day's shift pattern given by averaging all the penalty value of the consecutive two days appeared in the Table 1.

Table 1. Duty Patterns for the penalty F_{1i}.

p_{ij}	duty pattern for three consecutive days
0	DDD DDH DDM DHD DHH DHM DMS HDD HDM HHD HHH HMS SHH
1	DDS DSH DMH HDH HDS HHS HSH SHD MHD MHH MSH
2	DHS DSS HHM HSS HMH SHS SSH MDH MDS MHS MMH
5	DSD DSM DMD DMM HSD HSM HMD HMM SDD SDH SDS SDM SHM SSD SSS SSM SMD SMD SMS SMM MDD MDM MHM MSD MSS MSM MMD MMS MMM

It is not preferable for a night duty to be assigned for some nurse intensively. To suppress this situation, we define the following penalty function to prohibit *X* or more night duties to be assigned for consecutive *Y* days.

$$f_{2ij} = \begin{cases} \sum_{k=j}^{j+x-1} \dfrac{\max\left(N_{night/x}(i,j) - (y-1), 0 \right)}{N_{night/x}(i,j)} & j \in \mathbf{SHIFT} \\ 0 & \text{otherwise} \end{cases}, \tag{3}$$

$$F_{2i} = \sum_{j=1}^{D} f_{2ij} \quad , \tag{4}$$

where $N_{night/x}(i,j)$ denotes the number of night duties assigned for consecutive x days starting from j-(x-1)th day in the shift schedule of nurse i , y is defined as $y = xY/X$ and **SHIFT** denotes the set of days which shift is assigned.

In some hospitals, there are some cases to prohibit a specific duty pattern. If a shift pattern starting from j-th day of nurse i is prohibited pattern, penalty f_{3ij} is assigned to 1. We define a penalty function F_{3i} to implement such the prohibition as follows,

$$F_{3i} = \sum_{j=1}^{D} f_{3ij} \quad . \tag{5}$$

We define a penalty function on the number of duty days as the following equation,

$$H_2 = \sum_{i=1}^{M} \left(h_{21} F_{4i} + h_{22} F_{5i} + h_{23} F_{6i} \right), \tag{6}$$

where F_{4i} , F_{6i} and F_{6i} denote the following penalty functions about the number of duty days.

The number of duty days should be fairly assigned among nurses. A total nursing level falls, if many duty days are partially concentrated to particular nurses. We define the following penalty functions to suppress unevenness among nurses.

$$F_{4i} = \left| N_i^{\hom} - N_{\hom} \right|, \tag{7}$$

$$F_{5i} = \max\left(N_i^{sem} - N_{sem}, 0 \right) + \max\left(N_i^{mid} - N_{smid}, 0 \right), \tag{8}$$

where $N_i^{\hom}$, N_i^{sem} and N_i^{mid} denote the numbers of holidays, semi-night and midnight duties respectively assigned to nurse i for one month. $N_{\hom}$ denotes the number of Saturdays and Sundays on the current month. N_{sem} and N_{mid} denotes the limited numbers of semi-night and midnight duties, defined to four respectively in this paper.

If some nurses work many consecutive days, total nursing level falls. We define the following penalty function to restrain assigning more than X consecutive duty days.

$$f_{6ij} = \sum_{k=j}^{j+x-1} \frac{\max\left(N_{serial}(i,j) - (x-1), 0 \right)}{N_{serial}(i,j)}, \tag{9}$$

$$F_{6i} = \sum_{j=1}^{D} f_{6ij}, \tag{10}$$

where $N_{serial}(i,j)$ denotes the number of consecutive duty days starting from j-(x-1)th day in the shift schedule of nurse i and x is defined as follows,

$$x = \begin{cases} X & j \le D - X \\ y - 1 & \text{otherwise} \end{cases}, \tag{11}$$

where y denotes the number of the reminder days of the optimization period.

We define a penalty function on nursing level as the following equation,

$$H_3 = \sum_{j=1}^{D} \left(h_{31} F_{7j} + h_{32} F_{8j} + h_{33} F_{9j} \right), \tag{12}$$

where F_{7j}, F_{8j} and F_{9j} denote the following penalty functions about the nursing level.

In our algorithm, the number of the nurses in each duty time is secured by all means. However, the nursing level deteriorates if new face nurses are only assigned. The expert or more skilled nurses should be assigned to keep nursing level. The nursing level of each nurse is given by ten phases as shown in Table 2. We assume that the manager evaluates the nursing level in ten phases. We define the following penalty functions to perform the nursing level of each duty time.

$$F_{7j} = \max\left\{ L_j^{day} - \sum_i L(n_i), 0 \right\}, n_i \in M_j^{day}, \tag{13}$$

$$F_{8j} = \max\left\{ L_j^{sem} - \sum_i L(n_i), 0 \right\}, n_i \in M_j^{sem}, \tag{14}$$

$$F_{9j} = \max\left\{ L_j^{mid} - \sum_i L(n_i), 0 \right\}, n_i \in M_j^{mid} \quad , \tag{15}$$

where L_j^{day} , L_j^{sem} and L_j^{mid} denote the lowest nursing level at date j, and M_j^{day} , M_j^{sem} and M_j^{mid} denote sets of nurses assigned with day, semi night and mid night duty at date j. In this paper, we define L_j^{day} to 54 on week day, 33 on Saturday and 28 on Sunday, and L_j^{sem} and L_j^{mid} both to 16.

Table 2. Nursing Levels

nurse	m_1	m_2	m_3	m_4	m_5	M_6	m_7	m_8
level	10	9	9	8	8	8	8	8
nurse	m_9	m_{10}	m_{11}	m_{12}	m_{13}	m_{14}	m_{15}	m_{16}
level	8	7	7	7	7	7	6	6
nurse	m_{17}	m_{18}	m_{19}	m_{20}	m_{21}	m_{22}	m_{23}	
level	5	5	4	4	3	2	1	

We define a penalty function on nurse combination as the following equation,

$$H_4 = \sum_{j=1}^{D} \left(h_{41} F_{10j} + h_{42} F_{11j} + h_{43} F_{12j} \right), \tag{16}$$

where F_{10j}, F_{11j} and F_{12j} denote the following penalty functions about the nurse combination.

The manager also considers affinity between the nurses. Because of bad affinity between a certain nurses assigned to in the same time, there is the case that the nursing level deteriorates remarkably. We define a penalty function F_{10j}. When a pair of such the bad affinity is found in the shift schedule, penalty value 1 is added to the penalty function.

In the time of the midnight shift, only the nurse of few numbers of people is assigned to. The nursing level of the midnight shift deteriorates remarkably if the most of a nurse assigned to the

time are new faces. To restrain such the unfavorable situation, we define the following penalty function,

$$F_{11j} = \begin{cases} 0 & , N^{mid}_{j,new} < 2 \\ \sum_{i=0}^{N^{mid}_{j,new}-2} \left(N^{mid}_{j,new} - i - 1 \right) & , N^{mid}_{j,new} \geq 2 \end{cases} \quad (17)$$

where $N^{mid}_{j,new}$ denotes the number of new faces assigned to night duty on date j. In this paper, we define positions of nurse as shown by the Table 3. In this table, EX, BB and NF denote an expert, a backbone and a new face respectively.

Table 3. Positions of Nurses

m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
chief	head	head	EX	EX	EX	EX	EX
m_9	m_{10}	m_{11}	m_{12}	m_{13}	m_{14}	m_{15}	m_{16}
EX	EX	BB	BB	BB	BB	BB	BB
m_{17}	m_{18}	m_{19}	m_{20}	m_{21}	m_{22}	m_{23}	
BB	BB	NF	NF	NF	NF	NF	

In general, one or more expert or more skilled nurses must be assigned to day time and mid night duty. To restrain such an unfavorable situation, we define a penalty function, F_{12j}. If no expert or more skilled nurse is assigned to day time duty or mid night duty at date j, the function, F_{12j}, is increased with one point.

At the real hospital, the shift schedule which optimized before the beginning of the current month is often changed. Such a change of the schedule leads to the disproportion of the duty days. It causes the overwork of some nurses when such unexpected situation is ignored. Furthermore, there can be not only the fall of the nursing level but also the thing leading to a medical accident. To restrain such an unfavorable situation, we consider the re-optimization of the shift schedule of the remainder of the current month. First, we suppose that we have had the optimized shift schedule at the beginning of the current month. When several weeks have passed, we suppose that the shift schedule has been changed. We apply the CGA to re-optimize the shift schedule covering next four weeks including the remainder of the current month. However, we had better not change the shift schedule of the remainder of the current month if possible, because of the circumstances of each nurse. We define a penalty function, F_{13}, to optimize the shift schedule while having such a dilemma. The penalty function, F_{13}, performs the difference between the original schedule and the newly optimized schedule of the remainder of the current month.

Finally, we define a total penalty function of the shift schedule as follows,

$$E = \sum_{k=1}^{4} H_k + h_5 F_{13}. \quad (18)$$

The smaller value of the total penalty, E, means the better shift schedule.

2.3 Cooperative Genetic Algorithm

Basic algorithm of the CGA is shown in Fig.2 [5-7]. CGA applies the crossover operator to the population and searches so that a penalty of the whole population becomes small. The crossover operator selects a pair of parent individual from the population. Two child pairs are constituted by the two-point crossover. Taking back these child pairs to the original position of the parents, a temporal population is configured. The temporal population is evaluated by the penalty function E. These procedures are applied to one hundred parent pairs selected from the population. A population giving the best performance is selected for the next generation.

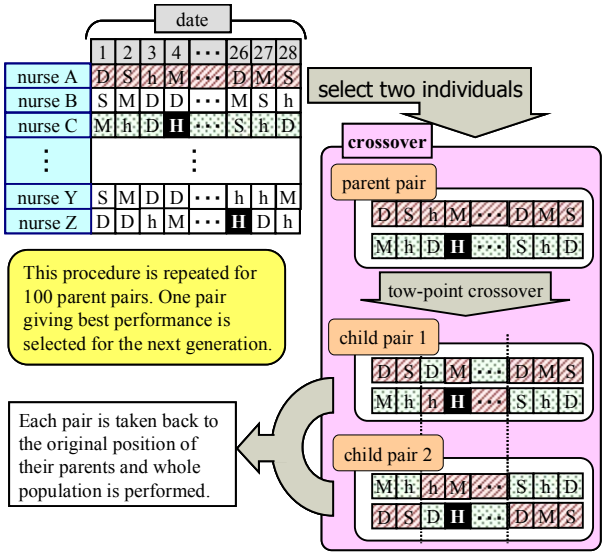


Figure 2. One generation cycle by using crossover operator.

2.4 Mutation Operator Depending on Optimization Speed

Since the nurse scheduling problem discussed in this paper is so difficult to solve, the optimization often stagnates only by using the crossover operator. The crossover operator is superior in ability for local search, but is unfavorable for global search. When the optimization stagnates for several generations, it is effective to forcibly give small change to the population. Therefore we have proposed a mutation operator activated depending on the optimization speed [8].

The operation of the mutation is shown in Fig. 3. The mutation operator randomly selects the duty date and selects two nurses. One of two is stochastically selected as giving the function F_1 big value. Another one is randomly selected. Then the mutation exchanges each other.

The mutation is activated depending on the optimization speed. The optimization speed is defined as follows.

$$A(g) = \frac{1}{N_g} \sum_{i=0}^{N_g - 1} E(g-i), \quad (19)$$

$$V(g) = A(g-1) - A(g), \quad (20)$$

where N_g denotes the number of generations from the last mutation to the current generation, $A(g)$ denotes the average value of the total penalty for N_g generations and $V(g)$ denotes the optimization speed. When the following condition,

$$V(g) < \varepsilon, \qquad (21)$$

is satisfied, the mutation operator is activated. The next mutation is not activated for G_g generations after the mutation. We call this the guard interval.

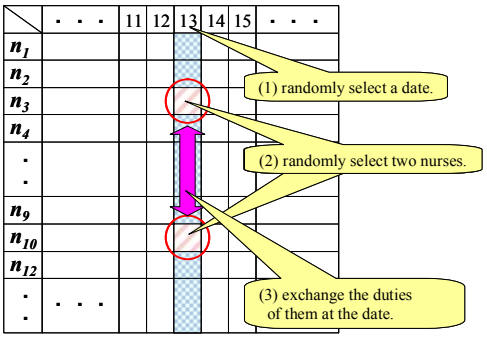


Figure 3. Mutation operator.

A population just before the last mutation is stored. Comparing with the last population and the population before this mutation, the population giving the smaller total penalty value is selected for the mutation. An example of the change of these values is shown in Fig. 4.

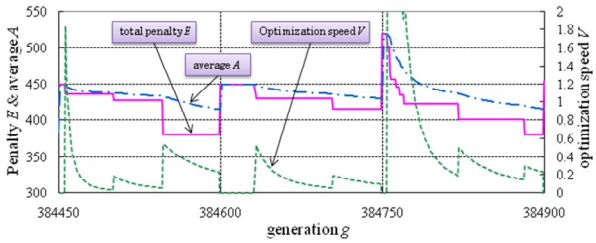


Figure 4. An example of the change of values of the total penalty, the average and the optimization speed. In this case, the mutation is activated at 384450th, 384600th and 384750th generations.

3. PRACTICAL EXPERIMENT

We have tried experiment of the nurse scheduling with practical data by the CGA. The number of the nurses is defined to twenty-three. We suppose that there have been several changes in the schedule in the past two weeks. The CGA re-optimizes the shift schedule for the coming four weeks. A part of the schedule on the first coming two weeks has been already given at the beginning of the current month. Therefore, difference, F_{13}, is performed to this part of the optimized schedule against the original schedule as shown in Fig. 5.

Fig. 6 shows the results of the optimization without the mutation and results with the mutation depending on the optimization speed. In one trial, the optimization is executed for 1,000,000 generations. Ten time of the optimization are executed for comparison. We have defined G_g as 50 from our experience. As shown in Fig. 6, the mutation effectively works for the optimization when the threshold is defined from 0.01 to 0.2.

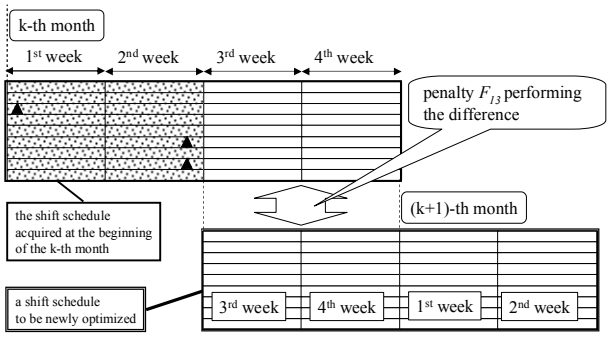


Figure 5. We expand the nurse scheduling to accept some changes in the past two weeks. When the two weeks have past, the coming four weeks are optimized to restrain inconvenience because of the changes.

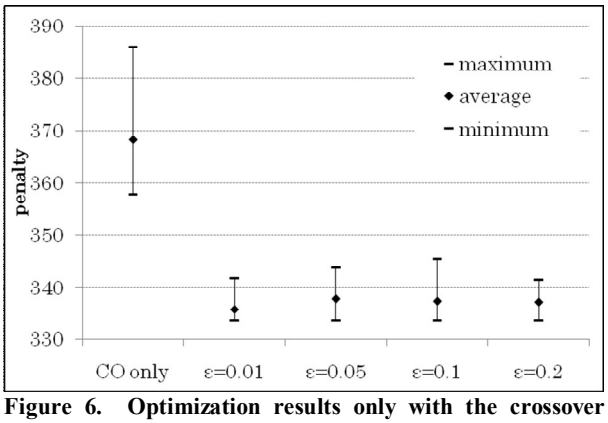


Figure 6. Optimization results only with the crossover operator (CO only) and optimization results with the mutation depending on the optimization speed with several threshold values.

4. IMPROVEMENT

4.1 Periodic Mutation Operator

The mutation with a threshold value except the range from 0.01 to 0.2 has brought unfavorable results. Besides, we must define the guard interval appropriately. In other words, we have to be careful to use the mutation depending on the optimization speed. So that, we propose a simple mutation operator activated periodically, where we call this the periodic mutation operator. A procedure flow of the periodic mutation is shown in Fig.7. The mutation operator is activated periodically every G_M generations. In the periodic mutation, the mutation period G_M is the only one parameter to define itself.

The optimization results by using the periodic mutation operator are shown in Fig.8. As shown in Fig.8, the periodic mutation yields results as almost equivalent as the conventional mutation operator. The mutation period is effective on wide range from 50 to 1000. This means the thing that does not have to mention a mutation period too much. Fig.9 shows processes of ten trials of the optimization using the periodic mutation operator with the mutation period, 700.

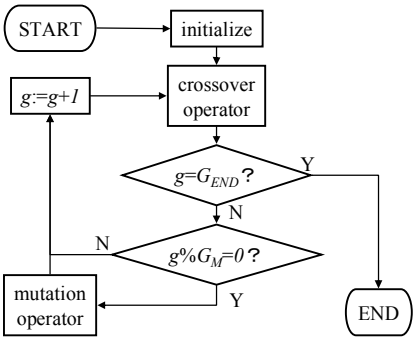


Figure 7. Procedure flow of CGA with the periodic mutation operator.

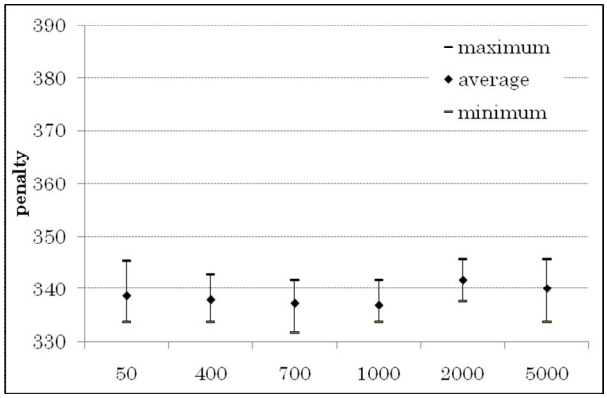


Figure 8. Optimization results by the periodic mutation operator with several mutation periods. We have tried to set the mutation period from 50 to 5000.

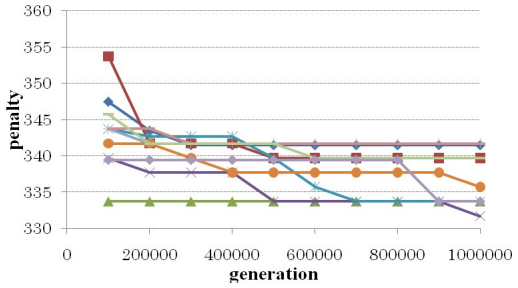


Figure 9. Ten trials of the optimization using the periodic mutation operator with the mutation period, 700.

4.2 Multi-Branched Mutation Operator

The nurse scheduling including such changes in the past part of the schedule, shown in the chapter 3, becomes very difficult. In our investigation, two hundreds thousands generations are enough to simply optimize a schedule of coming four weeks. However the optimization of the schedule of four weeks including such the changes requires more than one million generations. We have also investigated about optimization process in detail. The optimization sometimes stagnates even when the periodic mutation operator is applied. In the case of the most that the optimization stagnates, though the penalty F_1 has not been still decreased, other penalties has become almost zero. Therefore we need a technique to decrease the penalty F_1 when the optimization has been stagnated. In this paper, we propose a new mutation operator to improve this problem. The flow of this technique is shown in Fig.10. This technique is a multiplex divergence type mutation operator, where we call it Multi-Branched Mutation (MBM) operator.

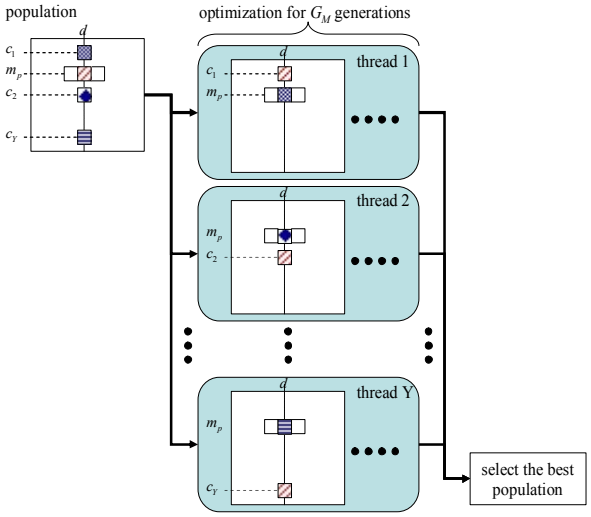


Figure 10. Multi-branched mutation operator.

The procedure of MBM operator is as follows. The number of nurses at each shift must be secured. Therefore, MBM determines a mutation point giving the largest value of the penalty F_1. MBM replaces the mutation point to other shift candidates of the day same as the mutation point, where we try three kinds of method to decide the shift candidates. The mutation point is given as a middle day of consecutive three days. If a fixed duty is not included on the middle day, the MBM decides it as a mutation position, (m_p, d). Else if fixed duty is included on the middle, the MBM checks it in order of the third day and the first day and decides either as a mutation position. If all three days contain fixed duty respectively, select a position giving the next bigger value of the penalty F_1. Figure 9 shows the case that the middle day does not contain such a fixed duty. Second, the MBM selects several candidate positions, (c_1, d), (c_2, d), ..., (c_X, d), on the same day, d, which does not contain such a fixed duty. We define the number of the candidate positions is N_{MBM}.Third, the MBM exchanges duty contents between the mutation position and all the candidate positions. By means of these procedures, we acquire new Y mutated populations. The CGA individually optimizes those mutated populations by using the crossover operator for G_M generations, where we call the individual optimization a thread. When all the threads have been completed, The MBM selects the best population for the next generation.

We investigate three kinds of method to decide the candidates. In the first method, MBM1, the mutation position (m_p, d) is replaced with the all candidates, (c_1, d), (c_2, d), ..., (c_X, d), and X new populations are forcibly generated. MBM1 must replace even a candidate giving no penalty value. This may cause a useless search, and it is not an efficient method. In the second

method, MBM2, the following sub-penalty Z of a candidate (c, d) are calculated,

$$Z(c,d) = \begin{cases} p_{cd} + f_{2cd} + f_{3cd} + f_{6cd} & (j < D-1) \\ p_{cD} + f_{2cd} + f_{3cd} + f_{6cd} & (\text{otherwise}) \end{cases} \quad (22)$$

MBM2 selects N_{MBM} candidates giving larger Z value for the MBM operation. In the third method, MBM3, N_{MBM} candidates giving larger F_{1i} value are selected for the MBM operation.

4.3 Parallel Processing of Nurse Scheduling

Nurse scheduling is very hard task even on a computer. In our investigation, an optimization for one million generations takes more than one hundred minutes. Actually around 10 time of optimization are necessary to get a good optimization result. This means enormous time and computational costs. If a user is not satisfied with those results, she/he retry several time of optimization. On the other hand, many computers are stopping at the late-night hospital. If parallel computation of the optimization is implemented by using these sleeping resources, we can achieve speedup of the nurse scheduling.

The MBM contains concurrency essentially. Those threads can be concurrently computed respectively, because optimization of these threads are independent each other. In other words the MBM is an operator suitable for parallel computation. We have implemented parallel computation of the CGA based on the MBM by using MPI™ technology. We already proposed several parallel processing techniques for the nurse scheduling [9]. The technique we propose in this paper is different from the conventional one.

Figs.11-13 show the optimization results by using MBM1, MBM2 and MBM3 with several generation ranges that the operation is applied. We have subtracted the number of generations executing MBM from the whole generations so that the evaluation number of times of the trial of MBM becomes the evaluation number of times same as the trials of the conventional technique. In other words, the evaluation number of times of the total penalty E is equal to when the conventional technique is applied. We have prepared two PCs equipping two CPUs respectively for the trial. In our implementation, one CPU is assigned to a server thread and an optimizing thread, and others are assigned to the optimizing thread.

In the case of MBM1, splendid schedule has been brought when MBM1 is applied to the middle stage of the whole generations. When MBM1 is applied to the final stage of the whole generations, the optimization results converge on the little value. When MBM1 is applied to whole generations, the optimization has unfavorably progressed. In the case of MBM2, splendid schedule has been brought when MBM2 is applied to the middle and final stages of the whole generations. Especially, when MBM2 is applied to the final stage of the optimization, the results converge on the splendid schedule. When MBM2 is applied to whole generations, the optimization has not progressed favorably nether. In the case of the MBM3, the optimization results converge on a good value even when MBM3 is applied to any stage of the whole generations. However, MBM3 has not brought such the splendid solution.

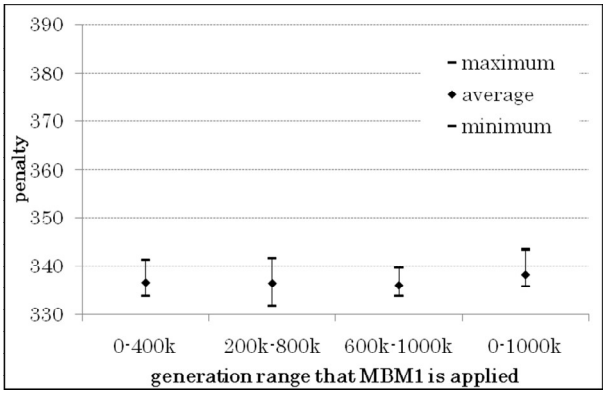


Figure 11. Optimization results by MBM1 with several generation ranges that MBM1 is applied.

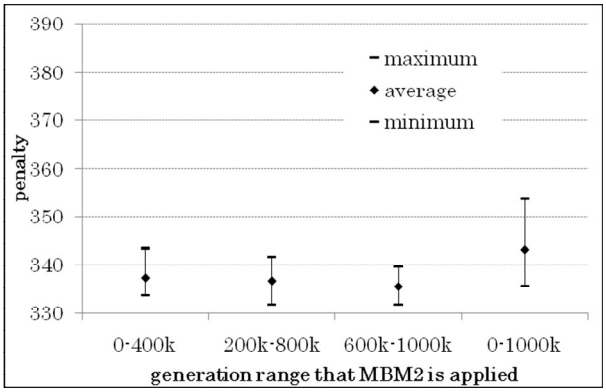


Figure 12. Optimization results by MBM2 with several generation ranges that MBM2 is applied.

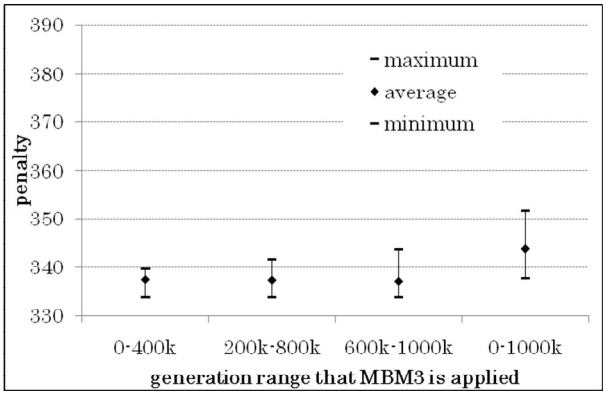


Figure 13. Optimization results by MBM3 with several generation ranges that MBM3 is applied.

5. CONCLUSION

This paper has shown a technique of nurse scheduling by using the CGA. The CGA effectively work to find a good schedule. However, the nurse must often attend unlike the original schedule in the real hospital, because of the circumstances of other nurse or an emergency and so on. We have discussed a case that the nurse schedule has been changed in the past weeks. To

re-optimize the changed schedule, we have defined a penalty function performing the difference. Such re-optimization of the nurse schedule becomes very difficult problem to CGA only with the crossover operator. Then we need new techniques to search for good schedule effectively. To solve this difficulty, we proposed the mutation operator activated depending on the optimization speed. Although this mutation operator effectively searches in the solution space of the nurse scheduling problem, several parameters must be defined. To improve this inconvenience, we have proposed the periodic mutation operator. The periodic mutation shows the performance that is about the same as conventional mutation. Besides, the periodic mutation needs only one parameter, the mutation period, which there is no need to mention too much. We have proposed MBM to optimize the nurse schedule more efficiently, and to shorten optimization time. Besides, MBM brings an effect of multiplexing a search process naturally. We have implemented the parallel processing of the nurse scheduling using MBM.

6. ACKNOWLEDGMENTS

This research work is supported by Tottori University Electronic Display Research Center (TEDREC).

7. REFERENCES

[1] Ikegami, A. *Algorithms for Nurse Scheduling*, Proc. of 11th Intelligent System Symposium, (2001), 477-480.

[2] Goto, T., Aze, H., Yamagishi, M., Hirota, M. and Fujii, S. "Application of GA, Neural Network and AI to Planning Problems," NHK Technical report (1993), No.144, 78-85.

[3] Kawanaka, S., Yamamoto Y., Yoshikawa D., Shinogi T. and Tsuruoka N. *Automatic Generation of Nurse Scheduling Table Using Genetic Algorithm*, Trans. on IEE Japan, vol.122-C, No.6 (2002), 1023-1032.

[4] Inoue, T., Furuhashi, T., Maeda H. and Takabane, M. *A Study on Interactive Nurse Scheduling Support System Using Bacterial Evolutionary Algorithm Enegine*, Trans. on IEE Japan, vol.122-C, No.10 (2002), 1803-1811.

[5] Itoga, T., Taniguchi N., Hoshino Y. and Kamei K. *An Improvement on Search Efficiency of Cooperative GA and Application on Nurse Scheduling Problem*, Proc. of 12th Intelligent System Symposium, (2003), 146-149.

[6] Ohki, M., Morimoto, A. and Miyake, K. *Nurse Scheduling by Using Cooperative GA with Efficient Mutation and Mountain-Climbing Operators*, 3rd International IEEE Conference Intelligent Systems (2006), 164-169.

[7] Ohki, M., Uneme, S., Hayashi, S., Ohkita, M. *Effective Genetic Operators of Cooperative Genetic Algorithm for Nurse Scheduling*, 4th International INSTICC Conference on Informatics in Control, Automation and Robotics (2007), 347-350.

[8] Uneme, S. Kawano, H., Ohki, M. *Nurse Scheduling by Cooperative GA with Variable Mutation Operator*, Proc. of 10th ICEIS (June 12-16, 2008), INSTICC, 249-252.

[9] Ohki, M., Uneme S., Kawano H. *Parallel Processing of Cooperative Genetic Algorithm for Nurse Scheduling*, Proc. of the 2008 4th International IEEE Conference Intelligent Systems (Sept. 6-8, 2008), vol.2, session 10, 36-41.

Communication-focussed approach for real-time neural simulation

Paul J Fox, Simon W Moore
University of Cambridge
Computer Laboratory
JJ Thompson Avenue
Cambridge
CB3 0FD
United Kingdom
{paul.fox, simon.moore}@cl.cam.ac.uk

ABSTRACT

Communication on- and off-chip now dominates the power and performance of modern electronic circuits. We propose the use of modern field programmable gate arrays (FPGAs) to investigate the communication properties of systems capable of simulating one billion neurons. Each FPGA provides gigabits of chip-to-chip communication bandwidth and on- and off-chip memory bandwidth. The FPGA structure allows us to control the allocation of this bandwidth in great detail allowing optimisations and analysis to be performed. We present our architectural explorations and initial findings.

1. INTRODUCTION

The structure of an individual neuron and its response to stimulus is well known, as are the complex functions performed by different regions of the human brain. While the amount of data processing that can be performed by a single neuron is very limited, the human brain as a whole is believed to be (for some tasks at least) the most complex computer in existence. How do we connect many simple components to form such a complex system?

Neurons can communicate with other neurons by producing action potentials. These travel via synaptic junctions, which can alter the magnitude of action potentials that pass through them and also delay their propagation. Complex functional units are formed from networks of these communicating neurons. However, the topology and communication characteristics required to perform useful functions are not well understood as these are difficult to observe with sufficient resolution in biological systems. To overcome this limitation, some neuroscientists design experimental neural networks with the aim of mimicking brain functions, and use simulation tools to help refine them.

This paper presents a neural network simulation architecture to support real-time simulation of one billion neurons. Real-time simulation of one billion neurons operating independently would be relatively straightforward, with the only requirement being sufficient available resources. However such a system would not produce useful results without also simulating the complex communication between neurons, and one billion neurons needs to communicate action potentials over a network of around one trillion synapses.

We begin by investigating the similarities and difference in the communication properties of biological neural networks and the electrical circuits that will be used to simulate them. This leads us to the conclusion that it is feasible to create a simulation system for large neural networks using a number of processing nodes, connected using a grid or torus topology.

In order to simulate biological neural networks using an electrical system we must select a suitable simulation algorithm. We have chosen Izhikevich's model [10], as we believe that it is best suited to our goal of real-time simulation of one billion neurons. This algorithm is analysed to determine its communication properties and the implications for the design of our neural network simulation architecture.

We then focus on the structure of a processing node, and in particular how it could be most efficiently implemented using a field programmable gate array (FPGA). The properties of an FPGA, and in particular the relative lack of on-chip memory compared to what could be implemented in a custom application specific integrated circuit (ASIC), require that we make careful use of on- and off-chip communication resources, and especially off-chip memory bandwidth, to create a system which can perform real-time simulation of the required number of neurons and their communication. We propose an architecture for a processing node that has a number of features that minimise the amount of communication required to perform the simulation, and also help to rationalise it so that it is better suited to the communication properties of the FPGAs used to implement processing nodes, the network used to connect them and their off-chip memory resources.

 ISBN 978-84-693-6141-2

We then discuss related work, focussing on how communication issues have been handled, and limitations that were encountered. Handling off-chip communication and making efficient use of off-chip memory bandwidth are found to be common obstacles to creating large simulations. Finally we discuss preliminary results and provide conclusions.

2. COMMUNICATION PROPERTIES

Comparing and contrasting the communication properties of biological neural networks and electrical circuits allows us to make informed choices when designing the architecture of a large-scale neural network simulator. There are some communication properties where biological neural networks and electrical circuits show significant similarities. We can exploit these similarities to make our simulation architecture more efficient. In other cases there are marked differences in communication properties. We may be able to exploit these differences to further increase the efficiency of our simulation architecture. In other cases we must look for ways of working around differing communication properties which would otherwise have an adverse effect.

Bassett and Greenfield [2] find that communication in biological neural networks exhibits a significant degree of locality and modularity, with neurons which are close in physical space networked into densely connected sets. Larger sets which perform more complex functions are then formed from smaller sets with relatively few additional connections. Hence the majority of communication between neurons covers short physical distances, with there being some medium distance communication, and much less over longer distances. However, it is still possible that any neuron could communicate an action potential with any other neuron, breaking these principles of modularity and locality. This type of communication will be referred to as "non-uniform" in the following discussion.

The locality and modularity communication properties of biological neural networks compare well with those of a simulation architecture composed of multiple processing nodes, connected using a grid or torus topology. The majority of communication will be kept on-chip, which is both power and resource efficient, particularly if we use an on-chip network [4]. The majority of off-chip communication will be nearest neighbour but we must still deal with the possibility of long distance communication.

Non-uniform communication is fundamentally unsuited to implementation using an FPGA, as it quickly consumes routing resources, and ultimately leads to functional units being rendered unusable as the switch nodes that they use to connect to other parts of the FPGA are saturated by routing other communication. Therefore we must find a way of transforming non-uniform communication into uniform communication. This can be achieved by implementing a packet switched network to communicate between processing nodes, with any non-uniform communication traversing as many processing nodes (each of which will contain a network router) and off-chip communication links as are necessary for it to reach its destination. However, replacing point-to-point wiring with a packet switched network will introduce communication latency, particularly if several processing nodes need to be traversed. Are biological neural networks tolerant of such latency?

While the time of arrival of action potentials at their target neurons is believed to be a significant factor of data processing by neural networks, the resolution requirements of this timing are believed to be of the order of one millisecond, particularly as biological neural networks lack a global clock. In contrast, the latency introduced by traversing the router in a processing node implemented using an FPGA would be no more than a few clock cycles, and the off-chip communication links used to connect processing nodes will have a bandwidth of many gigabits, particularly if the high-speed serial transceivers (which are integrated into many recently introduced FPGAs) are used. Hence there is ample time available for the simulation architecture to communicate an action potential from one simulated neuron to another, even if it is necessary for the communication to traverse several processing nodes and off-chip communication links.

However, we must consider how congested the on- and off-chip networks in the simulation architecture could get when we scale to simulating a biological neural network with up to one billion neurons. This depends on communication frequency and bandwidth requirements. Mead [15] observed that biological neurons communicate via action potentials at a frequency of no more than a few Hertz. The amount of "data" communicated by a single action potential is rather limited, consisting only of the magnitude of the action potential, its timing and (implicitly) the neuron from which it originated. However, the fan-out of a single action potential can be of the order of one thousand. While simple messages transmitting action potentials could be communicated both within a processing node and on off-chip communication links at a frequency of around 100MHz, creating one thousand separate messages in the simulation architecture in response to the creation of a single action potential by a simulated neuron would lead to massive network congestion, so we must find ways to keep the number of distinct messages required to communicate an action potential to all of its targets to a minimum.

We must also consider how the simulation architecture will determine the targets of action potentials. With one billion neurons and a fan-out of one thousand, we will need to store data describing at least one trillion action potential targets, with additional data required to control routing when communication traverses multiple processing nodes. This quantity of data exceeds the capacity of available on-chip memory, particularly in an FPGA where on-chip memory is limited to the amount of embedded Block RAM (BRAM). Hence we must store action potential target data in off-chip memory. Determining the destinations of action potential communications efficiently is critical to keeping a neural network simulation running in real-time. Hence, while modern FPGAs provide gigabits of off-chip memory bandwidth, we must nonetheless find ways of making accesses to data held in off-chip memory as efficient as possible.

In summary, a simulation architecture composed of multiple processing nodes implemented using FPGAs, connected in a grid or torus topology using high-speed serial communication links is well suited to simulating biological neural networks with around one billion neurons, connected by trillions of synapses. The challenges that we face are making efficient use of on- and off-chip networks to avoid excessive

congestion and saturation, and making efficient use of off-chip memory resources. However, before we can implement a simulation architecture we must select and analyse an algorithm which can be used to simulate biological neural networks.

3. SIMULATION ALGORITHM

We must select a suitable algorithm to simulate the generation of action potentials by biological neurons and their communication via synapses. If we are to achieve our goal of simulating one billion neurons in real-time, this algorithm must make efficient use of processing and communication resources.

We have chosen an algorithm proposed by Izhikevich [10] as we believe that it offers a good compromise between biological accuracy and computational efficiency. Izhikevich has compared his algorithm to many others that have been proposed [11], including that by Hodgkin and Huxley [8] which is used by many other neural network simulation systems.

Izhikevich's algorithm uses two floating-point equations, one quadratic and one linear, to simulate the behaviour of a single neuron. Equation 1 represents the membrane voltage of the neuron and Equation 2 the refractory voltage. These equations have two variables (v and u) and two constants (a and b). An additional variable, I represents the current incident action potential to the neuron. It is reset to zero after every evaluation of the equations for that neuron.

$$v' = 0.04v^2 + 5v + 140 - u + I \quad (1)$$

$$u' = a(bv - u) \quad (2)$$

An action potential is produced if $v >= 30mV$, in which case v and u are reset by Equations 3 and 4 respectively. This requires two more equations with two constants (c and d).

$$v' = c \quad (3)$$

$$u' = u + d \quad (4)$$

These equations are designed to operate in continuous time. Since continuous time simulation is not possible without using analogue circuits, we must approximate it by using discrete time simulation with sufficiently small time steps. Jin, Furber and Woods [12] propose a time step length of 1ms, which they believe samples the values of Equations 1 and 2 frequently enough to model the creation of action potentials by neurons accurately. Using discrete time simulation also allows us to time-multiplex resources on processing nodes amongst many simulated neurons, while still giving the appearance of real-time simulation. This significantly reduces resource requirements compared to implementing dedicated resources for every simulated neuron, which allows us to simulate more neurons per processing node than would otherwise be the case.

The described algorithm uses floating-point arithmetic, and hence requires floating-point computation hardware and data storage. Jin, Furber and Woods [12] proposes an implementation which uses only integer arithmetic, hence allowing the algorithm to be evaluated by a simple RISC processor with a short pipeline. This strategy makes efficient use of both power and resources, as well as minimising the time taken to evaluate the state of a single neuron. They also conclude that the variables and constants associated with Equations 1 to 4 can be stored as 16 bit signed integers without significant loss of precision or change in the observable generation of action potentials. Hence the data associated with each neuron can be stored in $16 \times 7/8 = 14$ bytes. Since off-chip memory systems tend to prefer dealing with data units which are powers of 2, this will be rounded up to 16 bytes. Therefore a simulation with one billion neurons will require 1.6×10^{10} bytes to store data relating to the neurons being simulated, which is approximately 16GB.

The effects of a neuron generating an action potential are specified by a sequence of three-tuples consisting of the target neuron, the magnitude of the action potential transmitted to the target neuron via a synapse (referred to as a "synaptic weight") and the propagation delay introduced by this synapse. The synaptic weight (which may be negative) should be added to the target neuron's current I value after the specified delay. Each neuron needs to be assigned a unique identifier so that action potentials can be routed to it from any other neuron in the system. As we wish to simulate one billion neurons, this identifier will need at least 30 bits, although it is advantageous to increase this to 32 bits both to allow for future expansion and since this is equal to the word size supported by most off-chip memory systems. We also need to consider the data required to represent the delay and synaptic weight. In [12], delays of up to 16ms are considered. With a time-step length of 1ms, 4 bits are required to represent the delay. It should be possible to represent the synaptic weight with 12 bits, and hence the total amount of data required to represent a synapse, including neuron identifier, delay and synaptic weight, is 6 bytes.

In a simulation with one trillion synapses, 6×10^{12} bytes will be required to represent them, which is approximately 6TB. Therefore the data storage requirements of the algorithm are dominated by the data required to represent synapses rather than that used to represent neurons. Even a small change in the amount of data required to represent a single synapse will have a significant effect on the total data requirements of the algorithm for simulated neural network of this size.

To avoid putting additional load on the off-chip network beyond that already placed on it by the communication of action potentials, each processing node will be restricted to accessing its own off-chip memory. The capacity and bandwidth requirements of this memory then depend on the number of neurons which will be simulated by a single processing node. If we assume that each processing node will simulate ten thousand neurons, then around 100MB of data will be required to represent both the simulated neurons and their synapses. This is a straightforward amount of memory to implement in a single processing node, however we must consider the bandwidth which will be required.

[12] suggests that Equations 1 and 2 are evaluated every 1ms so that the the values of v and u can be sampled with suf-

ficient accuracy. A simulated neuron could create an action potential whenever its equations are evaluated. However, Mead [15] suggests that biological neurons actually produce action potentials at a rate of just a "few hertz". Therefore we will suggest that action potentials can be produced by a neuron at an average rate of 10Hz. With ten thousand neurons simulated by one processing node and a fan-out of one thousand, the bandwidth requirements of one processing node will be of the order of 600MB/s for synaptic data and 160MB/s for the variables and parameters of Equations 1 to 4. Making allowance for the additional data required to perform routing of action potentials between processing nodes, the total bandwidth required is likely to be around 1GB/s. This is of the same order of magnitude as the bandwidth provided by modern off-chip memory controllers, and so we must make careful use of the available bandwidth if we wish to keep the simulation running in real-time.

We also need to consider the costs associated with evaluating the I value for each neuron, which is an input to Equation 1. With a fan-in of one thousand and a maximum firing rate of 10Hz we would expect to have to sum ten synaptic weights per neuron per 1ms cycle in addition to evaluating the Izhikevich equations themselves. Given that [12] was able to reduce these equations to ten ARM instructions, it can be seen that the summing of synaptic weights takes around half of the processing time of the algorithm.

Considering the data and processing time requirements of the algorithm together, it can be seen that the communication and application of synaptic weights takes a significantly greater proportion of total resources, particularly off-chip memory bandwidth, than evaluation of the Izhikevich equations. Our implementation must take account of this if we are to achieve our goal of the real-time simulation of one billion neurons connected by one trillion synapses.

4. IMPLEMENTATION

Based on our observations, we have developed the architecture for a processing node within a neural network simulation system shown in Figure 1. This has separate units to evaluate the Izhikevich equations, looking up and routing action potential messages and looking up and accumulation of action potential effects. An implementation has been produced using an Altera Stratix III FPGA on a Terasic DE3 evaluation board. Off-chip memory is provided using a DDR2 SDRAM interface, with the memory controller being a soft component provided by Altera and programmed into the FPGA.

The Izhikevich equations are evaluated in the Equation Processor, with a message sent to the Lookup Engine via the Input Router whenever an action potential is generated. The Input Router also takes in action potential messages that arrive from the off-chip communication links. Messages from the Input Router are then fed to the Lookup Engine. This determines what to do with each message. This can involve sending one or more messages on the off-chip communication links to neighbouring processing nodes. Alternatively messages can be sent to the Accumulator instructing it to apply a series of action potential effects to sets of simulated neurons hosted by this processing node, with the application of these effects being delayed as specified.

4.1 Equation Processor

The Equation Processor is responsible for evaluating the current state of the neurons that it hosts, using the Izhikevich equations presented in Section 3.

At present the processor is implemented using an Altera NIOS II soft core, with the simulation algorithm implemented in software written in C++. Since there is insufficient on-chip memory available in the FPGA, the parameters of all neurons and synapses are held in the off-chip SDRAM, which is accessed via the Avalon interconnect, which is a form of on-chip network, with the exception of the I values, which are held in the Accumulator, and also accessed via the Avalon interconnect.

The Equation processor is notified at the start of every 1ms time step by an external interval timer. This raises an interrupt in the processor, which triggers the software which evaluates the Izhikevich equations for each of the neurons which is hosted by the processing node (the I values). The equations for each neuron are evaluated sequentially, and provided that evaluation is completed for all neurons before the next time step begins, this part of the simulation will run in real-time. This requires that the total time taken to both evaluate the instructions used to implement the algorithm and to fetch the data that it requires does not exceed 1ms. The processor is theoretically capable of executing one instruction for every stage of its pipeline per clock cycle, but this will be reduced by pipeline stalls introduced when instructions and data which need to be fetched are not present in the processor's caches.

Instruction cache misses will be rare in the program used to implement the neural simulation algorithm since the core of the program is a vary small loop cycling over each of the neurons in turn, executing identical code for each neuron to evaluate Equations 1 and 2. The only difference in behaviour between neurons is a result of a neuron creating an action potential, which requires evaluating Equations 3 and 4 to reset the values of v and u and sending an action potential message to the Input Router.

Data for each of the neurons hosted by a processing node is held sequentially in the SDRAM. This leads to the SDRAM being accessed by the Equation Processor in a continuous band once per 1ms time step. Consequently there is very good spatial locality in these SDRAM accesses but almost no temporal locality, save that updated v and u values are written back to the off-chip SDRAM once they have been evaluated. Accesses to the SDRAM can thus be made efficient by using a data cache with prefetching and long burst reads. We aim for the data relating to the neurons whose equations are next to be evaluated to be present in the data cache, avoiding any need for the processor pipeline to stall.

The I values are held in an on-chip memory block within the Accumulator, which is the only significant block of on-chip memory in a processing node, except for the Equation Processor's caches. Accesses to this on-chip memory block over the Avalon interconnect will not lead to significant pipeline stalls as it should be possible to perform such access within one clock cycle.

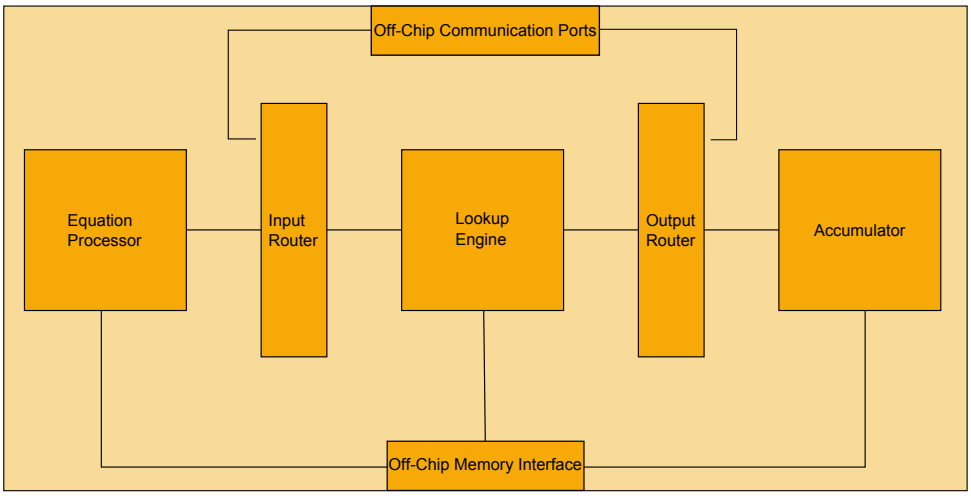


Figure 1: Processing node architecture

When a simulated neuron produces an action potential, the Equation Processor sends a message to the Input Router, which is then passed to the Lookup Engine. This message consists of a reference to a region of the off-chip SDRAM which is owned by the Lookup Engine. This reference is a 32 bit combination of the address and length of the region being referenced. The Equation Processor sends the message by writing to an address within the processor's memory map, which causes the data that is written to that address to be placed into a hardware FIFO in a custom communication component which is attached to the Avalon interconnect.

The number of simulated neurons that can be hosted by a single processing node can potentially be increased by using multiple Equation Processors, each of which will be responsible for evaluating the Izhikevich equations for a distinct set of simulated neurons. The DDR2 SDRAM is able to interleave memory accesses from multiple clients attached to the Avalon interconnect, so using multiple equation processors which share a single SDRAM will allow us to make more efficient use of the bandwidth to this off-chip memory.

4.2 Lookup Engine

The Lookup Engine determines the destination of action potentials, which may have originated either in a local Equation Processor or from the Lookup Engine in another processing node, via an off-chip communication link. It is implemented as a custom hardware block defined using the Bluespec SystemVerilog hardware description language [1], and connected to the off-chip SDRAM via the Avalon interconnect.

Action potential messages contain a memory reference (address and length) to a block of action potential routing instructions in a table held in off-chip SDRAM. A burst read is performed to fetch each of these routing instruction blocks from the off-chip SDRAM. The data which is returned consists of a series of routing instructions, which have two possible formats. The first format is a message containing the address of another processing node along with a memory reference to be sent to the Lookup Engine in that node. This message is passed to the Output Router and then routed to the appropriate off-chip communication link. The other format is a message to be passed to the Accumulator via the Output Router, containing a memory reference and a delay. The memory reference points to a block of action potential effects in a region of the off-chip SDRAM which is owned by the Accumulator. These affects should be applied after the delay specified in the message.

4.3 Accumulator

The Accumulator is responsible for storing and updating the values of the current incident action potential (I values) for each simulated neuron hosted by the processing node. It also implements the delays in communication of action potentials that are essential to the operation of neural networks. There is a single Accumulator per chip, regardless of how many Equation Processors are implemented.

When an Equation processor reads an I value, the Accumulator resets it to zero, ready to calculate a new I value for the next time step. At present the Accumulator's internal memory is memory-mapped into the address space of the Equation processor, and accessed over the Avalon interconnect. Given the frequency of such accesses, replacing this with a dedicated communication channel from each Equation Processor to the Accumulator brings a significant performance improvement.

Incoming messages from the Output Router contain a delay magnitude and a memory reference to a block of action potential effects in the off-chip SDRAM. Each of these effects contains a neuron identifier and an action potential to be summed with the current I value for that neuron. The delay magnitude indicates the number of time steps that must pass before the effects stored in the referenced effect block are applied. The Accumulator shares the Equation Processor's interval timer so that it can determine when a new time step begins.

Delays are implemented using sixteen hardware FIFOs, corresponding to the maximum magnitude of delay allowed by

the simulation algorithm. These FIFOs spill part of their contents to the off-chip SDRAM if they become full. The magnitude of delay that each FIFO currently represents is determined in a circular fashion relative to a pointer, which indicates the FIFO that currently represents a delay of zero, as proposed in [12]. The delay that each hardware FIFO represents can hence be updated at the start of a new time step simply by incrementing the pointer. When a message arrives from the Output Router, the reference to an action potential effect block is enqueued to the FIFO which represents the delay specified in the message.

When a new time step begins, one of the hardware FIFOs will have been promoted to represent a delay of zero. The Accumulator takes references to action potential effect blocks from this hardware FIFO and initiates a burst read to the off-chip SDRAM to retrieve them. A new burst read is initiated with a new effect block reference whenever permitted by the memory controller until the hardware FIFO is empty. The off-chip memory returns a stream of action potential effects, which are input into a simple pipeline. The first stage of the pipeline loads the current I value from the Accumulator's internal memory. The next stage performs an addition to apply the extra action potential and then the final stage stores the new I value back to the internal memory.

5. RELATED WORK

Neural network simulation is often performed in software, with one example of this approach being the Blue Brain project [14]. While software simulations allow for very large networks to be simulated very accurately (depending on the choice of simulation algorithm), this cannot be done in real-time for all but the smallest networks. Hence pure software simulation will not be discussed further here.

Much existing research into hardware neural network simulation has produced systems with hardware blocks which simulate neurons and synapses, connected together with groups of wires to transmit action potentials between them, an example being that by Indiveri, Chicca and Douglas [9]. The mapping from a biological neural network to the simulation hardware is very literal, in this case using analogue circuits, which makes the implementation easy to understand. However, even if digital circuits are used throughout, this approach suffers from particular difficulty with communicating action potentials between simulated neurons, as the wires used to facilitate this communication quickly consume the resources available, whether the implementation technology is an FPGA or a custom ASIC.

Without using some form of time-division multiplexing, the lack of available I/O pins compared to the number which would be required to communicate action potentials between simulated neurons placed on different chips makes expanding the simulation using multiple chips impossible. It is also difficult to alter the biological neural network which is being simulated, either the network is fixed by the ASIC implementation, or it is necessary to create and synthesise a new FPGA image. Upegui et al. [16] are able to simulate 50 neurons in a single FPGA, using run-time reconfigurable blocks to allow the communication properties of the simulated network to be altered without the time penalty of full reconfiguration.

A method of addressing the lack of expandability by representing action potentials as messages on links between chips was proposed by Boahen [3]. Neurons are simulated using analogue circuits and continue to communicate within a chip using point-to-point wiring, so this method is essentially providing a means of time-multiplexing the available wiring between chips. The number of neurons which can be simulated by a single chip remains limited as before, and the limitations of programmability remain.

Maguire et al. [13] propose a method to reduce the wiring complexity of a hardware neural network simulation, particularly when implemented using an FPGA, by storing the current state of simulated neurons in RAM and time-multiplexing both the wiring between simulated neurons and the simulated neurons themselves. A processor is used to control the scheduling of simulated neurons to resources, but it does not perform any calculation related to the neural simulation algorithm, which is still handled by dedicated hardware blocks. However, a combination of a more complex neural simulation algorithm than ours and what appears to be a lack of external memory bandwidth results in their simulation taking 90 minutes to simulate 1 second of activity in a neural network with 4200 neurons and 1964200 synapses.

Emery, Yakovlev and Chester [5] propose to replace the majority of the wiring between simulated neurons with an on-chip network, with point-to-point wiring remaining in small cells of connected neuron simulation blocks. A single simulation block is allocated per neuron, and it appears that the parameters of each neuron are configured into its allocated block before the simulation is run. The number of neurons which can be simulated by a single chip is limited, since the parameters of the simulated network must be stored in on-chip memory rather than more abundant off-chip memory. This limitation was identified by Hellmich et al. [7] and is shared by many other proposed neural network simulation systems.

Furber and Temple [6] are creating a neural network simulation architecture that has many similarities with that which we propose. Their SpiNNaker system is intended to consist of a number of processing elements, each of which contains 20 simple ARM processors. The properties of the simulated neurons are held in on-chip memory and the properties of their synapses and connectivity are held in off-chip SDRAM. Their simulation architecture uses significantly more on-chip memory than is possible with our proposed architecture as a custom ASIC implementation has been used. While this allows every parameter of the design of a processing element to be altered, significant design effort is required to create even a prototype system, and the costs of manufacturing these prototypes are orders of magnitude higher than creating prototype FPGA implementations. However, a custom ASIC implementation could ultimately lead to savings in power, resources and system cost if the ASICs that are produced are fabricated in sufficient volume, which will be the case if this architecture is ultimately extended to be capable of simulating one billion neurons in real-time.

6. PRELIMINARY RESULTS

The neural network simulation architecture described in Section 4 has been implemented within a single Altera Stratix III FPGA, hosted on a Terasic DE3 evaluation board. We have successfully simulated 3200 neurons in real-time using the Izhikevich neural network simulation algorithm described in Section 3. The average fan-out is between five and ten, and the system clock frequency is 200MHz. As expected, the load on the off-chip memory is significant, but analysis of the proportion of time that the off-chip memory is idle suggests that it should be possible to host at least four times as many neurons in a single FPGA.

The evaluation board used has proved to be unsuited to networking multiple boards in a grid or torus topology. While it possesses four high-speed I/O connectors which are designed to allow multiple boards to be connected together, only three of them are usable if the system being implemented also makes use of the DDR2 SDRAM connector. A new DE4 evaluation board has recently been released, which supports several additional communication protocols, and most significantly the Stratix IV FPGA used has integrated high-speed serial transceivers. We plan to be able to implement our neural network simulation architecture using this platform, which should allow us to gain a better appreciation of the communication properties of the architecture in greater detail and to simulate much larger neural networks with larger numbers of synapses in real-time.

7. CONCLUSION

The main issues facing the creation of an architecture for the efficient, real-time simulation of large neural networks are communicating action potentials between simulated neurons and storing and communicating the properties of the simulated neurons and their connectivity.

The communication properties of biological neural networks, particularly their locality of communication allow us to use a neural network simulation architecture composed of multiple processing nodes, connected using a grid or torus topology. By representing action potentials as messages, and using a packet-switched network to communicate them to their destination, we are able to use an on- and off-chip network to communicate action potentials while making efficient use of energy and resources.

The amount of data required to represent the properties and connectivity of a large neural network, particularly its synapses, means that this data cannot be stored in on-chip memory and must therefore be stored in off-chip memory. The neural network simulation algorithm requires a significant amount of bandwidth between off-chip memory and the hardware used to evaluate the simulation algorithm. While the bandwidth required is less than that theoretically available with modern FPGAs and memory technology, we must make careful use of the available bandwidth if we are to maintain a real-time simulation.

We believe that the architecture we have presented addresses these communication issues, and will ultimately achieve the goal of the real-time simulation of a biological neural network with one billion neurons and one trillion synapses, while making efficient use of power and resources.

8. REFERENCES

[1] www.bluespec.com.

[2] D. S. Bassett, D. L. Greenfield, A. Meyer-Lindenberg, D. R. Weinberger, S. W. Moore, and E. T. Bullmore. Efficient physical embedding of topologically complex information processing networks in brains and computer circuits. *PLoS Comput Biol*, 6(4):e1000748, 04 2010.

[3] K. Boahen. Point-to-point connectivity between neuromorphic chips using address events. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 47(5):416–434, May 2000.

[4] W. J. Dally and B. Towles. Route packets, not wires: on-chip inteconnection networks. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 684–689, New York, NY, USA, 2001. ACM.

[5] R. Emery, A. Yakovlev, and G. Chester. Connection-centric network for spiking neural networks. In *NOCS '09: Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 144–152, Washington, DC, USA, 2009. IEEE Computer Society.

[6] S. Furber and S. Temple. Neural systems engineering. *Journal of The Royal Society Interface*, 4(13):193–206, 2007.

[7] H. Hellmich, M. Geike, P. Griep, P. Mahr, M. Rafanelli, and H. Klar. Emulation engine for spiking neurons and adaptive synaptic weights. In *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, volume 5, pages 3261–3266 vol. 5, July-4 Aug. 2005.

[8] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952.

[9] G. Indiveri, E. Chicca, and R. Douglas. A vlsi array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity. *Neural Networks, IEEE Transactions on*, 17(1):211–221, Jan. 2006.

[10] E. Izhikevich. Simple model of spiking neurons. *Neural Networks, IEEE Transactions on*, 14(6):1569–1572, Nov. 2003.

[11] E. Izhikevich. Which model to use for cortical spiking neurons? *Neural Networks, IEEE Transactions on*, 15(5):1063–1070, Sept. 2004.

[12] X. Jin, S. Furber, and J. Woods. Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 2812 –2819, 1-8 2008.

[13] L. P. Maguire, T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin. Challenges for large-scale implementations of spiking neural networks on fpgas. *Neurocomput.*, 71(1-3):13–29, 2007.

[14] H. Markram. The blue brain project. *Nat Rev Neurosci*, 7(2):153–160, February 2006.

[15] C. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629 –1636, oct 1990.

[16] A. Upegui, C. A. PeÒa-Reyes, and E. Sanchez. An FPGA platform for on-line topology exploration of spiking neural networks. *Microprocessors and Microsystems*, 29(5):211 – 223, 2005.

P systems simulations on massively parallel architectures *

José M. Cecilia
Computer Engineering and Technology Department
University of Murcia
30100 Murcia, Spain
chema@ditec.um.es

José M. García
Computer Engineering and Technology Department
University of Murcia
30100 Murcia, Spain
jmgarcia@ditec.um.es

Ginés D. Guerrero
Computer Engineering and Technology Department
University of Murcia
30100 Murcia, Spain
gines.guerrero@ditec.um.es

Miguel A. Martínez–del–Amor
Computer Science and Artificial Intelligence Dept.
University of Seville
41012 Seville, Spain
mdelamor@us.es

Mario J. Pérez–Jiménez
Computer Science and Artificial Intelligence Dept.
University of Seville
41012 Seville, Spain
marper@us.es

Manuel Ujaldón
Computer Architecture Department
University of Malaga
29071 Malaga, Spain
ujaldon@uma.es

ABSTRACT

Membrane Computing is an emergent research area studying the behaviour of living cells to define bio-inspired computing devices, also called P systems. Such devices provide polynomial time solutions to NP-complete problems by trading time for space. The efficient simulation of P systems poses challenges in three different aspects: an intrinsic massively parallelism of P systems, an exponential computational workspace, and a non-intensive floating point nature. In this paper, we analyze the simulation of a family of recognizer P systems with active membranes that solves the Satisfiability (SAT) problem in linear time on three different architectures: a shared memory system, a distributed memory system, and a set of Graphics Processing Units (GPUs). For an efficient handling of the exponential workspace created by the P systems computation, we enable different data policies on those architectures to increase memory bandwidth and exploit data locality through tiling. Parallelism inherent to the target P system is also managed on each architecture to demonstrate that GPUs offer a valid alternative for high-performance computing at a considerably lower cost: Considering the largest problem size we were able to run on the three parallel platforms involving four processors, execution times were 20049.70 ms. using OpenMP on the shared memory multiprocessor, 4954.03 ms. using MPI on the distributed memory multiprocessor and 565.56 ms. using CUDA in our four GPUs, which results in speed factors of 35.44x and 8.75x, respectively.

Keywords

Multicore, Manycore, GPUs, P systems, SAT problem, High Performance Computing

*This work was supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, by the Spanish MEC and MICINN under project TIN2009-13192, by the FEDER funds of the European Community, under grants CSD2006-00046 and TIN2009-14475-C04, and by the Junta of Andalucia of Spain under projects P06-TIC02109 and P08-TIC04200. We would also like to acknowledge the support of the Centro de Supercomputación de la Región de Murcia where our experiments were conducted.

1. INTRODUCTION

Parallel computing architectures have brought dramatic changes to mainstream computing. This trend is accelerating as the end of the development of hardware following Moore's law looms on the horizon. The number of transistors per die are no longer relying on a single chip design, but being partitioned among a bunch of simpler cores. Multicore CPUs are holding a dozen of cores, and manycore GPUs gather a myriad of stream processors. These components are being combined to build heterogeneous parallel computers offering a wide spectrum of high speed processing functions. Major hurdles to exploit this raw power are the PCI express bus to communicate the CPU and the GPU as they do not share the memory space, and also their different parallel programming approaches and paradigms. These problems amplify when we move to heterogeneous clusters.

This paper explores this complex situation for a challenging application which requires (1) a dynamic handling of memory space and (2) an exponential workspace growing as our code increases the number of variables involved to run the simulation. Our simulation characterizes Membrane Computing, an emergent research area which studies the behaviour of living cells to define bio-inspired computing devices, also called P systems. These devices provide polynomial time solutions to NP-complete problems by trading time for space. This is inspired by the capability of cells to produce an exponential number of new membranes in polynomial time, through *mitosis* and *autopeosis* processes.

Currently, we lack of a feasible biological implementation, either *in vivo* or *in vitro*, of P systems. The only way to analyze and execute these devices is on silicon-based architectures which are limited by the physical laws. Although some simulators and software applications have been derived [8, 7], most of these simulators were developed for sequential architectures using languages such as Java, CLIPS, Prolog or C, where performance is hardly compromised.

Section 2 of this article introduces Membrane Computing and describes the behaviour of this biologically inspired way of computation, focusing on computational devices called P systems to solve the Satisfiability (`SAT`) problem. This behaviour is simulated on different architectures, namely, a shared-memory architecture (HP Superdome), a distributed-

 ISBN 978-84-693-6141-2

memory machine (cluster of HP Blades), and finally a set of Nvidia Tesla GPUs. Section 3 describes the parallelism which can be extracted from a P system simulation with active membranes, and once this is learnt, we demonstrate in Section 4 how GPUs can accommodate two levels of parallelism in its computational model versus a single level on shared and distributed memory systems.

The nature of P system computation creates an exponential workspace leading to polynomial time solutions for NP-complete problems. Section 5 analyzes different data policies to increase the memory bandwidth, and also to take advantage of the data locality on each architecture by providing a blocking/tiling algorithm. We also get a glimpse of the memory limitations on each system to simulate larger datasets and benchmarks. The GPU memory is very limited compared to the other alternatives, and the only way to include more GPU memory is actually adding more GPUs to the system. Finally, Section 6 highlights the main ideas presented, and provides some directions for future work.

2. BACKGROUND AND RELATED WORK

2.1 Membrane computing and P systems

Gh. Păun introduced Membrane Computing in 1998 [12], and since then, this bio-inspired computing paradigm has attracted research activities within Natural Computing. The model starts with the assumption that processes taking place in the compartmental structure of a living cell can be interpreted as computations. Devices of this model are called P systems, which consist of a cell-like membrane structure, where compartments allocate multisets of objects, that is, sets of objects with multiplicities associated to the elements.

P systems have several syntactic elements (see Figure 1): First, a membrane structure consisting of a hierarchical arrangement of membranes embedded in a skin membrane, which delimits the internal region of the P system from the environment. Second, delimiting regions or compartments where multisets of objects (corresponding to chemical substances) and sets of evolution rules (corresponding to reaction rules) are placed. Every membrane has associated an unchangeable label, and depending on the P system model, it may also contain a charge or polarization that can be modified during the computation. Besides, P systems possess two valuable features: inherent parallelism and non-determinism.

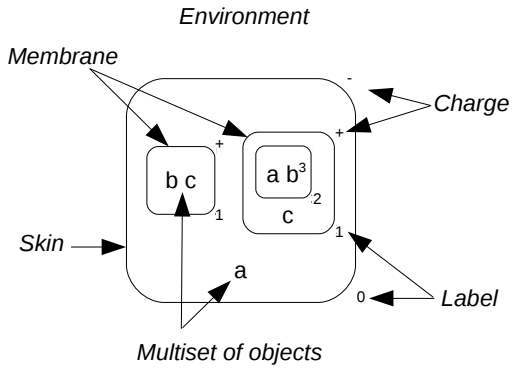


Figure 1: The structure of a P system.

A P system computation is a (finite or infinite) sequence of instantaneous transitions between configurations. The computation starts with an initial configuration of the system, where the input data of a given problem is encoded. The transition from one configuration to the next is performed by applying rules to the objects inside the regions. This process iterates until no more rules can be applied to the existing objects and membranes.

Note that P systems exhibit two levels of parallelism: one for each region (the rules are applied in a parallel way), and another one for the system (all regions evolve concurrently). The objects inside the membranes evolve according to given rules in a synchronous, parallel, and non-deterministic way.

The two level parallelism and non-determinism can be used to solve NP-complete problems in polynomial time, reducing this from an exponential time, but at the expense of using an exponential workspace of membranes and objects which is created in polynomial (often linear) time.

Up to date, there have not been *in vivo* nor *in vitro* implementations of P systems, and researchers have focused on simulators developed in silicon whose initial versions were targeted to sequential platforms [7, 8]. From this departure point, the main challenge for the simulations of P systems in general is to find the right platform to exploit massively the parallelism inherent to the definition of P systems.

In this respect, several efforts have been done implementing this massively parallelism on parallel architectures. For instance, Alonso et al. [3] proposed a circuit implementation for the class of transition P systems. Moreover, Nguyen et al. [9] proposed an implementation of transition P systems in FPGAs, providing several levels of parallelism, one at rule level and other at region level, releasing a software framework for Membrane Computing called Reconfig-P. A generic simulator on GPUs for a family of recognizer P system with active membranes was presented in [5], showing that the double level of parallelism exposed by GPUs represents a valid alternative to simulate P systems.

2.2 The Satisfiability (SAT) problem

Propositional Satisfiability problem (`SAT`) was the first known **NP**-complete problem, as proven by Stephen Cook in 1971 [6]. In computational logic, `SAT` is a decision problem aimed to determine, for a formula of the propositional calculus in Conjunctive Normal Form (CNF), if there is an assignment of truth values to its variables for which that formula evaluates to true. This is of paramount importance in many computer science areas, including theory, algorithmic, artificial intelligence, hardware design, electronic design automation, and verification.

We assume a formula to be in CNF when it is a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a variable or its negation (the negation of an expression can be reduced to negated variables by De Morgan's laws). For example, a_1 is a positive literal and $\neg a_2$ is a negative literal.

Considering a CNF formula φ with n variables $(x_1...x_n)$ and m clauses $(C_1...C_m)$, the time spent by all known deterministic algorithms to solve the `SAT` problem is exponential depending of the size of the input $(max\{m, n\})$ in the worst case. With the help of membrane systems, we are able to find the solution at linear time but at the expense of creating an exponential workspace.

The P system simulation algorithm to solve the SAT problem is based on the P system computation described in [13], which can be summarized as the following list of stages:

1. **Generation.** Membranes are structured within a rooted tree with a single branch. The root node is the *skin membrane*, and the second node is called *internal membrane*. All possible truth assignments to the variables are generated by using division rules, and they are encoded in the internal membranes by executing step by step the set of P system rules already described in [13]. In this way, 2^n internal membranes are created such that each one encodes a truth assignment to the variables of the formula.

2. **Synchronization.** The objects encoding a true clause (a partial solution to the CNF formula) are unified in the membrane.

3. **Check out.** The goal here is to determine how many (and which) clauses are *true* in every internal membrane (that is, by the assignment that represents).

4. **Output.** Internal membranes encoding a solution send an object to the skin. If the skin has such object from some membrane, the object Yes is sent to the environment. Otherwise, the object No is sent.

Algorithm 1 summarizes the sequential code based on previous stages. First, *Generation* and *Synchronization* are the stages creating an exponential workspace of membranes in a synchronous way, and also unifying the objects that codify a partial solution. Both stages are executed in the same function, which is referred to as *Generation* from now on. Note that each membrane runs in parallel at each iteration of *Generation*, but a global synchronization is required by different iterations.

Once the workspace is created, the *Check out* and *Output* stages are performed. First, they determine the clauses being true in every internal membrane, and then they check whether there is a solution for the SAT problem. Hereafter, we combine these two stages into a joint *CheckOut* function.

Algorithm 1 The sequential pseudocode of the P system simulation algorithm for the SAT problem with n variables.

Require: $n \geq 0$
{Start Generation and Synchronization stages}
repeat
Generation
until n
{Start Check out and Output stages}
CheckOut

The specific simulation of the family of P systems that solves `SAT` for a single GPU is analyzed in [4], where problems to carry out the theoretical simulation of P systems on GPUs are depicted, and some heuristics to accelerate its computation are provided.

3. THE PARALLEL VERSION FOR THE P SYSTEM SOLVING THE SAT PROBLEM

The P system for the SAT problem gathers all computational features of the recognizer P systems with active membranes [11]. Among them, we highlight the theoretical double level of parallelism and non-determinism that makes P systems a computational tool to solve NP-complete problems in polynomial time.

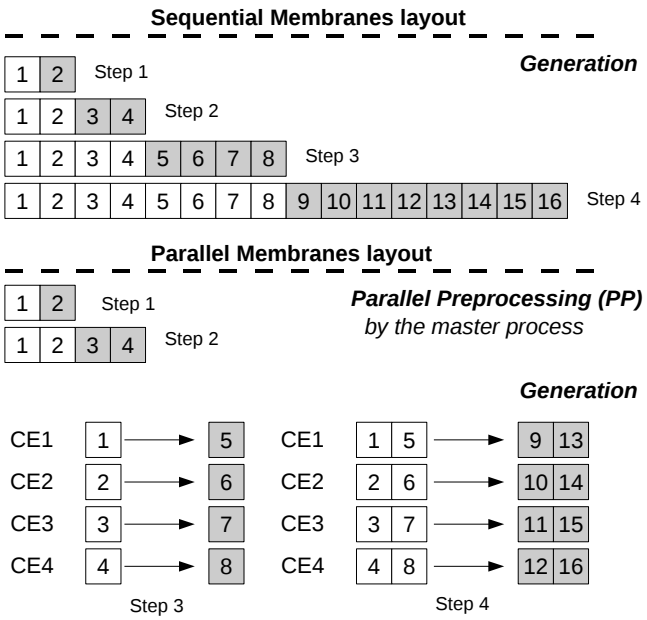


Figure 2: Sequential and parallel membranes generation on four Compute Elements (CE). The *Parallel Preprocessing (PP)* is required to set up the parallel execution.

The first level of parallelism for the SAT P system is found among membranes, that is, by executing each membrane in parallel along the computation. The second level of parallelism is found within each membrane. That way, the first level is coarse-grained and can be characterized by an inter-task parallelism and exploited by the number of processors available in the parallel system, whereas the second level of parallelism is fine-grained and intra-task to be exploited by the number of cores within each processor, either on multi- or many-core architectures.

The membrane parallelism is showed in Figure 2. It shows the execution of the *Generation* function for the SAT P system in a sequential as well as a parallel architecture with four Compute Elements (CE). In a parallel architecture, a set of membranes is initially created by the master process, whose size is equal to the number of CEs available during the execution. Then, a membrane is sent to each CE by the master processor. This step is called *Parallel Preprocessing (PP)*, and it is developed just before the *Generation* starts the computation on each CE. This CE is represented by a processor (die) on each hardware platform, which can later be eventually decomposed into multi- or many-cores when exploiting intra-task parallelism.

Furthermore, Figure 2 shows that each membrane is always generated by the same membrane and also in the same computational step on every architecture. For instance, membrane two is always generated by membrane one in the first computational step, membrane three is always generated by membrane one in the second step, and so on. Finally, each node sends the partial response back to the master in order to produce the final result of the P system.

Figure 3 shows the second level of P system parallelism (that internal to membranes). Once the initial data has arrived to the CE after the *Parallel Preprocessing* step, it

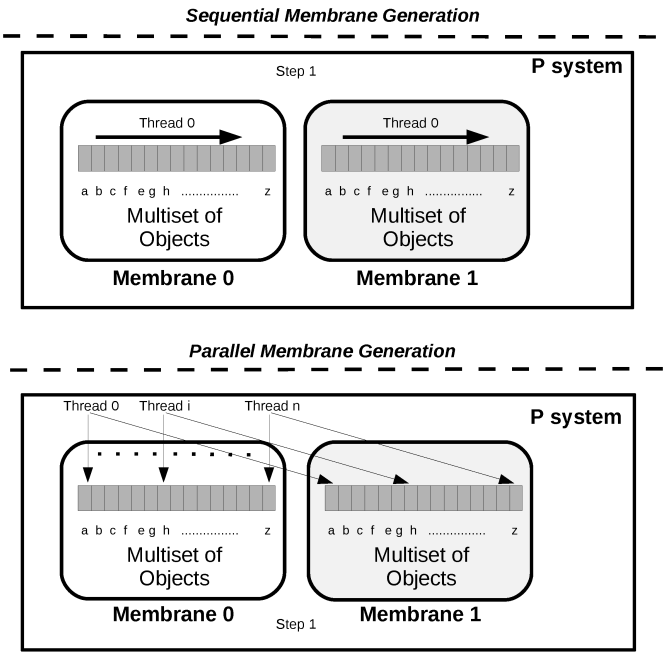


Figure 3: Sequential and parallel execution when creating the exponential workload.

starts the computation according to algorithm 1, and applying the P system rules for the SAT problem depicted in [13]. Then, resources on each CE can be exploited at its peak to cooperate for speeding up the computation of the *Generation* and *CheckOut* functions. This resources are essentially hardware cores on shared memory, distributed memory and GPU platforms, but only GPUs are manycore which can handle this level of parallelism at large scale using hundreds of streaming processors (see Table 1).

4. DATA POLICIES DESCRIPTION

Our P system simulator for the SAT problem organizes data depending on the features of the underlying architecture. We now describe those data policies.

4.1 The shared memory implementation

The simulator was implemented on the shared memory system using OpenMP [2]. Figure 4 shows the first data layout used by our simulator. The shared memory space is equally distributed among the n processes considered, and the master process performs the *Parallel Preprocessing* step by creating as many membranes as number of processors are involved in the computation. Membranes are placed at the beginning of the memory space assigned to each process (see gray squares in Figure 4). Now, the *Generation* step is carried out by each individual process, writing the information on its own memory fragment. Once the membranes workspace has been created by the *Generation* stage, the *CheckOut* stage follows, where membranes are read again by processes to eventually produce the system response.

This data policy does not take advantage of data locality when the *Generation* and *CheckOut* stages are performed, thus producing many caches misses (in particular, read misses) that hit the simulator performance. Locality was improved through a block-based data layout as shown in Figure 5.

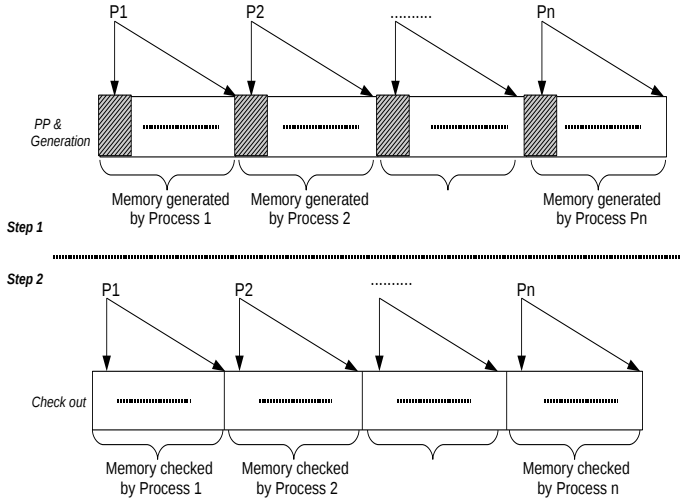


Figure 4: Initial data placement for our shared memory implementation.

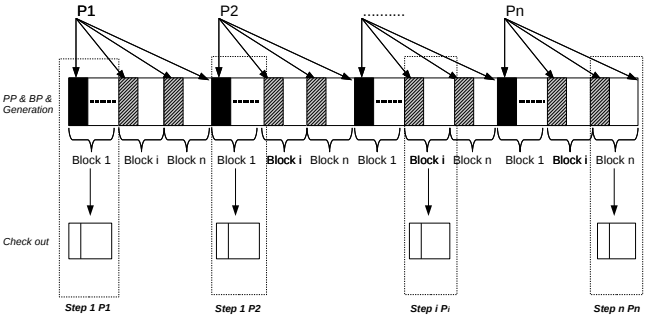


Figure 5: The shared memory implementation for n processes using our block-based data layout.

Again, the master process starts with the *Parallel Preprocessing* step, which generates as many membranes as processes (represented by black squares in Figure 5). But now each process performs a local preprocessing step (called *Block Preprocessing*, (BP)) on its own memory space before starting the *Generation* stage itself. *Block Preprocessing* pursues a tiling or blocking execution between different stages of the simulation. Each process creates as many membranes as number of blocks, placing them at the beginning of each block position (represented by gray squares in Figure 5). Then, the *Generation* stage only creates *blocksize* membranes before the *CheckOut* stage starts. Once a block has been checked by the *CheckOut* stage, processes start again the *Generation* on the following block it has assigned to.

The block-based data policy increases the time required by preprocessing, including a new BP stage, but a shorter data block can be placed in higher levels of the memory hierarchy, which benefits from data locality. However, there is a trade-off between preprocessing computation (PP and BP) and the data locality benefits for the *Generation* and *CheckOut* stages, being affected by the block size chosen.

4.2 The distributed memory implementation

The P system simulator for the SAT problem on the distributed memory system was programmed using MPI [1].

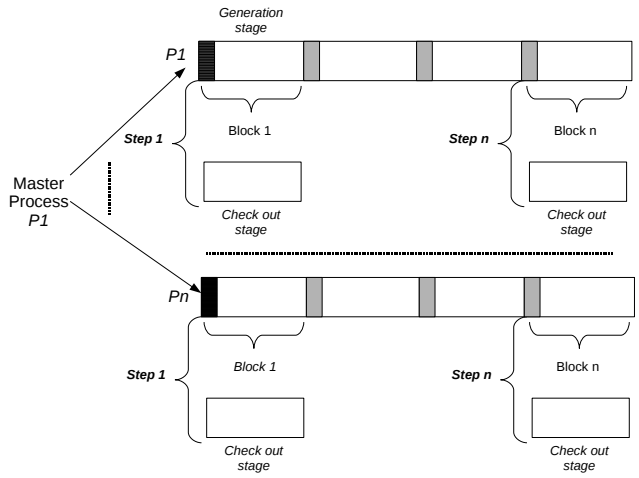


Figure 6: P system simulation on a distributed memory architecture.

Again, we compare here a preliminary non-blocking version with an enhanced version based on a blocking data policy.

In this case, each process allocates memory on its own and private memory space. The master process also performs the *Parallel Preprocessing* step, creating as many membranes as number of processors are involved in the computation. Then, membranes are sent to processors by using the MPI `Scatter` instruction.

Once the initial data arrives to each node, the P system computation was developed as in the shared memory case. For the non-blocking data policy, the *Generation* is fully performed before the *CheckOut* starts its computations. For the block-based data policy, the *Block Preprocessing* is required for a blocking or tiling execution. Figure 6 shows the data layout for the block-based data policy. Finally, a reduction is applied using the MPI `Reduce` instruction to end up with the system answer.

4.3 Implementation on GPUs

The simulator sets a CUDA thread block for each membrane and a CUDA thread per object (or set of objects) in the multiset.

This time, the first attempt for the SAT P system simulation on GPUs, the *Generation* stage, is encoded as a CUDA kernel, and it starts right after the *Parallel Preprocessing* step. Once membranes have been generated, the *CheckOut* stage starts its execution. Each thread block loads a membrane from global memory, and then each thread checks the rules associated with this stage. Finally, each block returns whether its associated membrane makes true the CNF formula or not. For these stages, all threads within a CUDA thread block cooperate with coalesced access to device memory (threads of the same warp access the same memory segment either for reading or writing).

Blocking can also be exploited on GPUs, taking advantage of the on-chip shared memory by using tiles with the aim of increasing the bandwidth to device memory (see Figure 7). The simulation has to perform the *Block Preprocessing* step, which is implemented through a CUDA kernel where a set of membranes are partially created, placing them apart from each other at a block size distance.

An additional kernel is created this time at the end of the

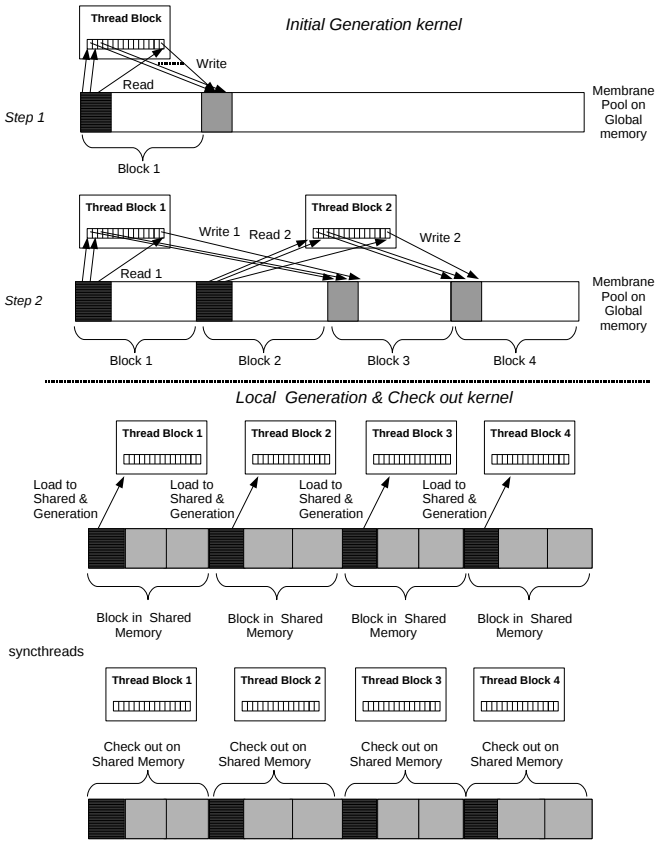


Figure 7: P system simulation on a single GPU.

simulation. This kernel performs the *Generation* locally to each block, followed by the *CheckOut* stage. Each thread on a thread block cooperates for an efficient load from global memory to shared memory of the initial membrane generated by the *Block Preprocessing* step (represented by black squares in Figure 7). Then, the *Generation* stage interacts with shared memory, saving expensive loads/writes from/to global memory which are around 400 times slower.

Finally, the *CheckOut* stage is performed over the data stored in shared memory after a block-level synchronization. This checks whether a clause makes true the CNF formula, and writes its result into device memory.

Figure 8 shows the data policy used by the simulation of the P system for the SAT problem on a GPU-based platform. This simulator arranges data according to the "best practices" existing at this moment for CUDA enabled devices with CUDA Compute Capabilities (C.C.C.) 1.3 [10]. Nevertheless, those guidelines are mainly focused on arithmetic intensive applications on a single GPU. It remains to be seen whether they are valid on architectures like GPU-based clusters with a much higher degree of parallelism.

Within a GPU-based cluster, GPUs cannot interact with each other, and a CPU process has to be created to monitor each GPU independently. Note that this does not force us to use parallelism at CPU core level, as we have exactly four CPUs in our system which can individually host each of the required processes. This way, our three implementations lack of using the multithread capabilities of CPU cores.

Figure 8 shows how the master thread creates four CPU threads (CPU context) to invoke the execution on each GPU

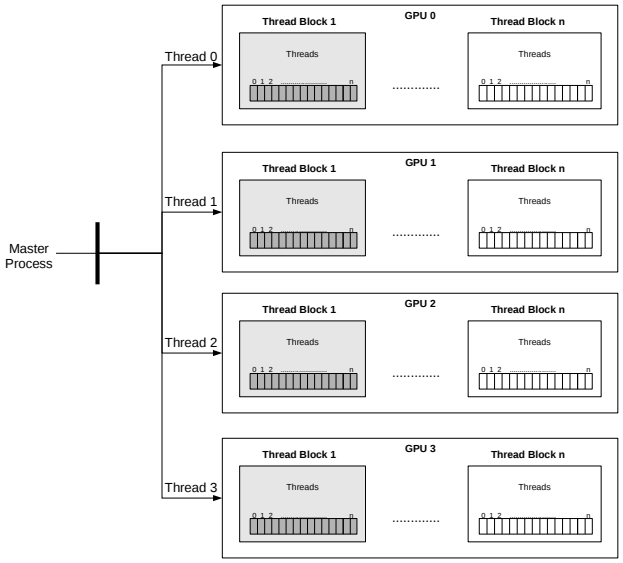


Figure 8: Data policy on a set of four GPUs.

and manage its resources (i.e allocate device memory, move data to/from the GPU, and so on). Resources created on each CPU thread are not accessible by any other thread, and there is no explicit initialization function for the runtime API [10], which makes hard to measure time in a reliable manner, particularly on multi-GPU environments.

For the GPU case, the master process performs the *Parallel Preprocessing* step as usual, generating as many membranes as GPUs are involved in the simulation, and performing the assignment.

At a starting point, the simulation barely exploits GPU resources because the computation begins with a single CUDA thread block (which represents the membrane generated by the *Parallel Preprocessing* step). However, the number of CUDA thread blocks grows exponentially in the *Generation* stage along with the number of membranes, and GPU resources are fully utilized at early stages of the simulation. Another alternative consists of creating a larger set of initial membranes in the *Parallel Preprocessing* step to fulfill that GPU resources are occupied right from the beginning, but we have tested that this initial low usage of GPU resources has a negligible impact, even on tiny benchmarks.

5. PERFORMANCE EVALUATION

This section evaluates our P systems implementations in three different platforms. Hardware features are summarized in Tables 1 and 2.

The shared memory platform is a HP Integrity Superdome SX2000 endowed with 64 CPUs, Intel Itanium 2 dual-core Montvale (16 Kbytes L1, 256 Kbytes L2, 18 Mbytes L3). Total DRAM memory available is 1.5 Tbytes and interconnection network is a 4x DDR Infiniband.

The distributed memory system is a HP BladeSystem which contains up to 102 nodes and each node is a dual-socket, each containing a quad-core Intel Xeon E5450 (Nehalem with a 12 Mbytes L2 cache). DRAM memory capacity for the whole system is 1072 Gbytes. Interconnection network is also a 4x DDR Infiniband.

Finally, our GPU-based platform include a four-socket,

Table 1: CUDA and hardware features for the Tesla C1060 GPU used within our GPU-based platform.

Feature	Limitation
Multiprocessors (SM)	30
Streaming processors / SM	8
Total number of streaming processors	240
32-bit registers / SM	16384
Shared memory / SM	16 KB
Threads / SM	1024
Threads / Block	512
Threads / Warp	32
Device (video) memory available	4 GB

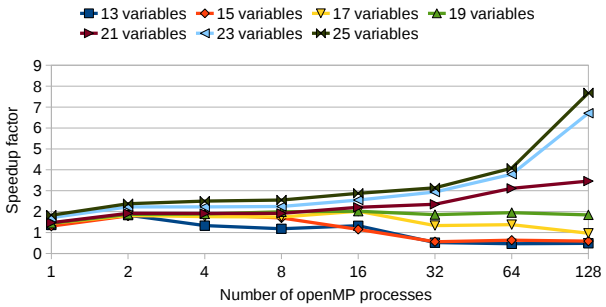


Figure 9: Speed up factor achieved by the blocking algorithm when varying the number of variables.

quad-core Intel Xeon E5530 (Nehalem with a 8 Mbytes L2 cache), which acts as a host machine for our four Nvidia Tesla C1060 GPUs whose details are shown in Table 1.

Data policies and simulation performance are evaluated on each architecture under a set of benchmarks generated by the WinSAT program [14]. WinSAT can generate random SAT problems in DIMACS CNF format file by configuring several parameters: the number of variables *(n)*, the number of clauses *(m)* and the number of literals per clause *(k)*.

The number of membranes in our P system depends on the number of CNF variables, n ($Membranes = 2^n$). We vary this parameter from $n = 13$ variables (2^{13} membranes) to $n = 25$ variables (2^{25} membranes), whereas the number of literals ($l = m \times k$) is kept constant ($l = 256$ for benchmarks with $n < 22$ and $l = 200$ for benchmarks with $n \geq 22$). Doing so, we reduce memory requirements so that more benchmarks can be simulated on the GPU-based system. Memory requirements for each benchmark can be calculated according to Equation 1.

$$Size = 2^n(membranes) \times l(objects) \times 4(uint)\ bytes \quad (1)$$

5.1 The shared memory platform

A performance comparison between the blocking and non-blocking algorithm for 64 membranes per block is shown in Figure 9. The blocking technique increases performance either with the problem size (i.e. the number of variables in the CNF formula for the SAT problem) or the number of computational processes (OpenMP processes created). The former is needed to hide the *Preprocessing time* (PP and BP), and the latter involves the memory coherence protocol: The network traffic in shared memory systems goes up

Table 2: Summary of hardware features for the architectures used during our experimental survey.

	Shared memory	Distributed memory	GPU-based
Hardware platform	Hewlett-Packard Integrity Superdome SX2000	Hewlett-Packard Blade System	4 Intel Xeon E5530 CPU (plus 4 Tesla GPUs described in Table 1)
Number of nodes	1	102	1
CPU sockets per node	64	2	4
CPU cores per socket	2	4	4
CPU cores and speed	128 @ 1.6 GHz	816 @ 3 GHz	16 @ 2.4 GHz
Main memory (DRAM)	1536 GB	1072 GB	16 GB. (+ 16 GB. video memory)
Programming model	OpenMP (+ Linux 64 bits)	MPI (+ Linux 64 bits)	CUDA (+ Linux 64 bits)
Compiler	icc Intel 11.1	HP MPI 02.03.01	nvcc Nvidia 2.3

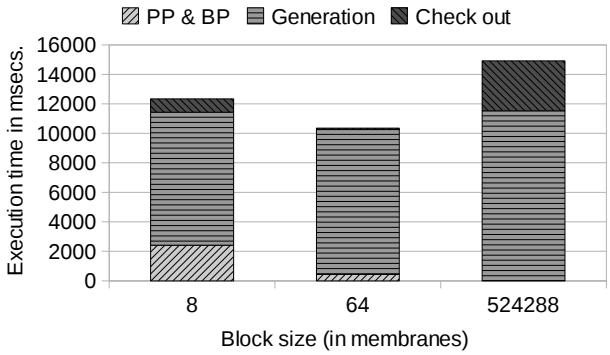


Figure 10: Breakdown for the total execution time using 8 processes for a SAT problem composed of $n = 23$ variables and $l = 200$ literals.

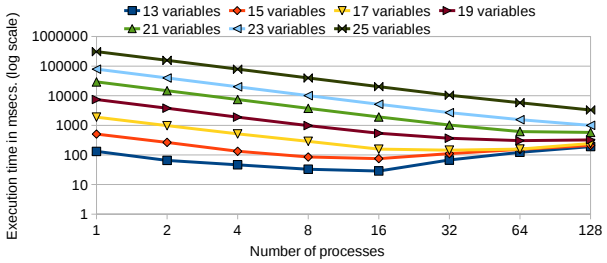


Figure 11: OpenMP code performance varying the number of variables for the block-based version.

with the number of cores, but the blocking technique takes advantage of the local data stored on each node to reduce the communications burden versus the non-blocking version.

We now present some results about the simulation performance of the SAT P system, depending on the block size for the block-based data layout in our shared memory system. Figure 10 shows the breakdown for the total execution time in the three main functions performed by the OpenMP simulation, depending on the block size used by the blocking technique. We have checked many different block sizes to find the best configuration, but for the sake of simplicity Figure 10 only shows three of them for the benchmark with $n = 23$ variables: the largest block size configuration, the shortest one, and finally the one scoring peak performance.

The largest block size ($2^{19} membranes/block$, up to 420 Mbytes according to Eq. 1) is the most time-consuming configuration. The *Preprocessing* (PP and BP preprocessing) step is the least time-consuming for this configuration because only a few initial membranes are required in advance, but the *Generation* and *CheckOut* stages are heavier than in the other two configurations. *CheckOut* starts reading the first membrane right after the $2^{19} membranes$ of a block are generated by each process. Since the L3 cache size for the processor in our shared memory architecture system is 18 Mbytes, many read and write cache misses occur in those stages, affecting the overall simulation performance.

Similarly, the smallest block size ($2^3 membranes/block$) shows the highest *Preprocessing* time. Although the *Generation* and *CheckOut* stages behave much better on cache misses, the simulation finds its best configuration for 2^6 membranes (50 Kbytes) per block. This is the turning point between *Preprocessing* time and *Cache misses* (write and read misses) for this architecture.

Finally, Figure 11 shows the execution time (in a log scale) for the SAT P system simulation with the best configuration under the blocking technique. We executed several benchmarks varying the number of variables of the SAT problem, and also varied the number of OpenMP processes involved in the computation for each benchmark in order to study the scalability of the system.

$$t_{total} = t_{prepro} + t_{cpu} + t_{overhead} \quad (2)$$

The total execution time is given by the equation 2. The first parameter (t_{prepro}) is the preprocessing time spent by the master process to create the initial set of membranes to be distributed among remaining processors; this is *Parallel Preprocessing* plus the preprocessing time needed by each process to prepare the blocking execution (that is, *Block Preprocessing*). It depends on two values: the number of processes and the block size. The second parameter (t_{cpu}) concerns the processing time taken by each node, and depends on the benchmark size. Finally, the last parameter ($t_{overhead}$) is the extra overhead added to the OpenMP execution time (i.e. synchronizations, loop scheduling, communications among processors, resource sharing, etc...). This parameter increases widely with the number of OpenMP processes.

Figure 11 shows that the scalability of the system grows with the problem size, as processing time (see equation 2) predominates over remaining parameters as long as the problem size increases. This scalability gets reduced on smaller benchmarks.

Note that this version only exploits the intra-task parallelism (that is, among membranes). Remaining stages for the simulation are sequentially performed on each node.

5.2 The distributed memory platform

In this case, the maximum speed up obtained by the best configuration for the blocking technique algorithm reaches up to 2x versus the non-blocking alternative, with this peak

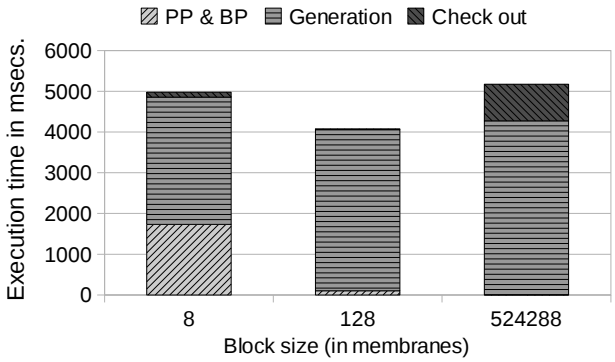


Figure 12: Breakdown of the total execution time using 8 MPI cores with $n = 23$ variables and $l = 200$ literals.

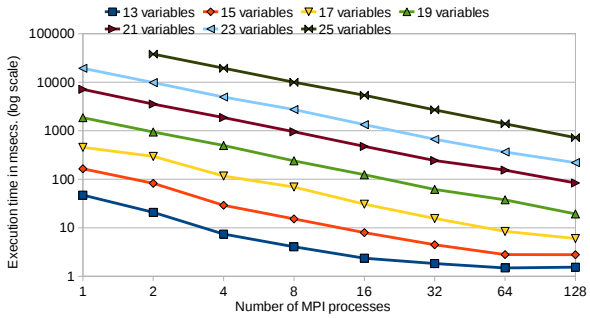


Figure 13: MPI code performance varying the number of variables.

reached for the case of the $n = 25$ variables benchmark. Memory banks are independent on this platform, so the blocking algorithm takes advantage of data locality to improve memory bandwidth.

Regarding the optimal data block size, Figure 12 shows the breakdown of the total execution time for the three main functions performed by the MPI simulation for the benchmark with $n = 23$ variables. Again, Figure 12 shows only the largest, shortest, and best performance block size configurations. The optimal case here corresponds to 2^7 membranes per block (100 Kbytes per block).

Figure 13 shows the execution time (in a log scale) for the MPI code, taking the best configuration blocking technique and varying the number of variables of the SAT problem and the number of MPI processes. The total execution time can also be given by the equation 2. Minor differences are seen based on the architectural features of each system, with the overhead being influenced by communications among processors. Data sent to each processor by the master is a single membrane, and the result returned by each node is just a boolean, saying whether or not a solution is found.

Figure 13 reveals that the system scalability improves again with the problem size, but it scales much better than in the OpenMP case. Results on a single core are missing for the largest benchmark (that of $n = 25$ variables), because the memory available on a single node is not enough to run the simulation (the benchmark allocates up to 26 Gbytes and the maximum memory per node is 16 Gbytes).

Note that this version does not exploit the inter-task parallelism either: Each membrane is sent to a node and simulations are executed sequentially on that node.

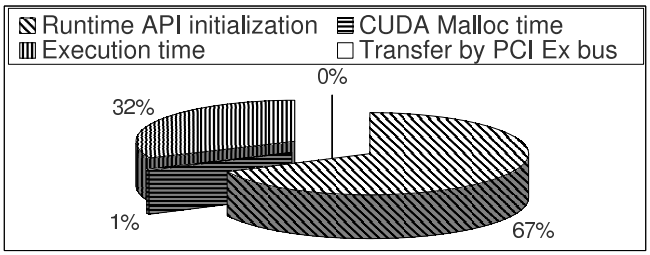


Figure 14: Breakdown of the total execution time in a single GPU with $n = 22$ variables.

5.3 The set of four GPUs

In this case, the tiling technique obtains up to 1.75x speed up factor versus the non-tiling counterpart.

Figure 14 shows the breakdown of the total execution time for a single GPU executing the benchmark with $n = 22$ variables and using a tiling version. It shows that the 67% of total execution time is spent by the runtime API initialization on average, and only 32% corresponds to the actual execution time. Data transfers are not that important here, and lose the leadership shown on previous platforms.

The runtime API initialization penalty is not usually considered when timing GPU applications because it is not stable between different executions nor related to the actual GPU computation. But in our case it represents two thirds of the total execution time, so we decided to include it within GPU times even though it goes against its performance over the other two architectures.

First of all, we evaluate the impact of the data block size. Figure 15 shows the breakdown of the total execution time for the two main kernels performed by the GPU simulation. The block size is now limited by the on-chip shared memory space (16 Kbytes for Tesla C1060). Simulations are tested for two, four and eight membranes per block, reaching the best performance for the last case.

The number of global memory accesses and the number of iterations in the *Block Preprocessing* kernel intrinsically depends on block size. In particular, eight membranes per block require half of the memory accesses and iterations as compared to the four membranes per block configuration, which, similarly, cut down to a half those required by the two membranes per block case. Figure 15 reflects this fact.

Likewise, memory accesses in the *Generation* and *Check-Out* stages are reduced in a similar way as long as the block size increases. However, the GPU resource occupancy worsens for the eight membranes per block case, because the shared memory usage per block prevents from allocating more than one block per SM. As a result, the overall improvement is just 14% over the four membranes per block configuration.

Figure 16 shows the performance for the tiling version of the GPU simulator with eight membranes per block, and varying the problem size. The number of GPUs is also increased to study the scalability for the system.

In a multi GPU environment, Figure 16 shows a linear speed up along with the number of GPUs. This is expected as the computational workload is evenly distributed on GPUs. Furthermore, there is more room on each GPU memory space, so higher workloads may be executed. Figure 16 shows that a single GPU cannot execute the benchmark with n=23 variables, whose memory requirements are 6400

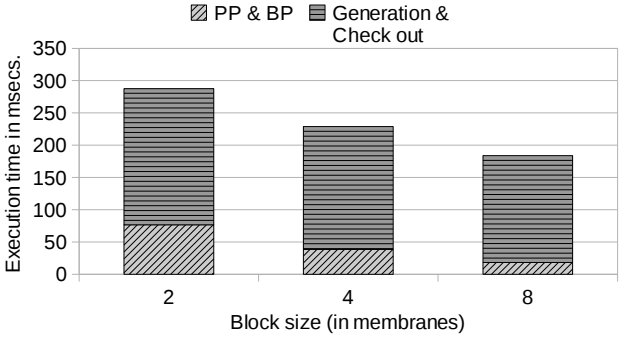


Figure 15: Breakdown for the execution time on a single GPU with $n = 23$ variables and $l = 200$ literals.

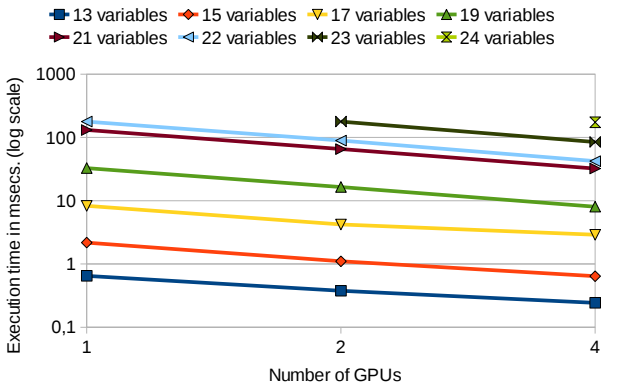


Figure 16: CUDA performance when varying the number of variables (on y axis) and GPUs (x axis).

Mbytes. Similarly, two GPUs cannot execute the benchmark with n=24, with its size reaching 12800 Mbytes. At this point, we recall that a P systems simulation creates an exponential workspace to obtain polynomial time solutions for NP-complete problems. So, the benchmark composed of n=25 variables consumes 25.6 Gbytes, which again becomes unfeasible on four GPUs.

Times in Figure 16 do not account for overheads like *initial* and *final* data transfers between CPU and GPU, GPU memory allocation, and CUDA runtime initialization, which may be significant in practice. *Parallel Preprocessing* time spent to arrange the execution on multiple GPUs is also ignored, though this time is negligible as the simulation creates just four membranes on a four GPUs configuration.

GPUs improve significantly the device memory bandwidth through shared memory usage, which is explicitly used by the CUDA programmer. This way, one can control the number of accesses and the way to access on memory bounded applications like ours. Even though the small size of the shared memory decreases GPU occupancy, the benefit of reducing the number of accesses to device memory is much higher and this strategy is widely rewarded.

5.4 Overall comparison

Figure 17 summarizes the performance for all our implementations. For the smallest benchmark, GPU performance gets severely affected by initialization overheads, but this is quickly amortized as we increase the problem size. The situation reverses for larger benchmarks, reaching its peak for $n = 23$ variables, where the problem size only fits into two or four GPUs, and that is the reason why the time on a single GPU is missing. With the last run for $n = 25$ variables requiring 25.6 Gbytes, we were unable to execute it on GPUs even considering together the video memory of our four GPUs.

Considering the largest problem size and amount of parallelism we were able to expose on the three parallel platforms for a fair comparison ($n = 23$ variables and four processors), execution times were 20049.70 msec. using OpenMP on the shared memory multiprocessor, 4954.03 msecs. using MPI on the distributed memory multiprocessor and 565.56 msecs. using CUDA in our set of four GPUs. Consequently, the speed-up we attain with our set of GPUs reaches 8.75x versus the distributed memory system and 35.44x versus the shared memory platform for a much cheaper high-performance alternative.

6. CONCLUSIONS

In this article, we have described the simulation of a family of recognizer P systems with active membranes, solving the satisfiability (SAT) problem, on three different parallel architectures base on shared memory, distributed memory and a set of GPUs. We have also used three different programming models: OpenMP, MPI and CUDA, respectively.

Our data placement analysis reveals that blocking increases the bandwidth in all targeted systems by taking advantage of data locality, but performance varies depending of the memory architecture and the way to manage it. We also dedicate some efforts to reduce the cost of preprocessing steps required for applying this technique on each platform.

The blocking technique improves the parallel efficiency of the shared memory architecture, but the OpenMP simulator reaches the lowest performance as the pressure on shared resources increases with the number of processors. On the positive side, this was the only platform where we were able to execute all benchmarks due to higher memory availability.

The distributed memory system exhibits good scalability with the number of processors, which can be partially explained by the low number of communications required by our simulations.

GPUs constitute the best platform to simulate P systems for SAT in terms of execution time. The two levels of parallelism that P systems exhibit, one at region level and another one at system level, were exploited by our GPU implementation to reach speed-up factors around 10x versus distributed memory and around 40x versus shared memory when four processors are used on a given platform.

For the future, the newest generation of many-core GPU architectures, Nvidia Fermi, enhances the GPU with memory resources to develop general purpose applications and more sophisticated models of P systems. Moreover, the combination of cloud computing and heterogeneous systems can be an alternative for increasing the memory size without sacrificing performance at all.

Alternative models of P systems which could be used to computationally replicate biological systems within the framework of population and systems biology (i.e., probabilistic/stochastic models) are well positioned to be successfully simulated on multi- and many-core systems due to its arithmetic intensity and large number of iterations required to adjust the model. A high-performance implementation of those simulation models looks promising on GPUs and we have provided some guidelines to succeed by using CUDA.

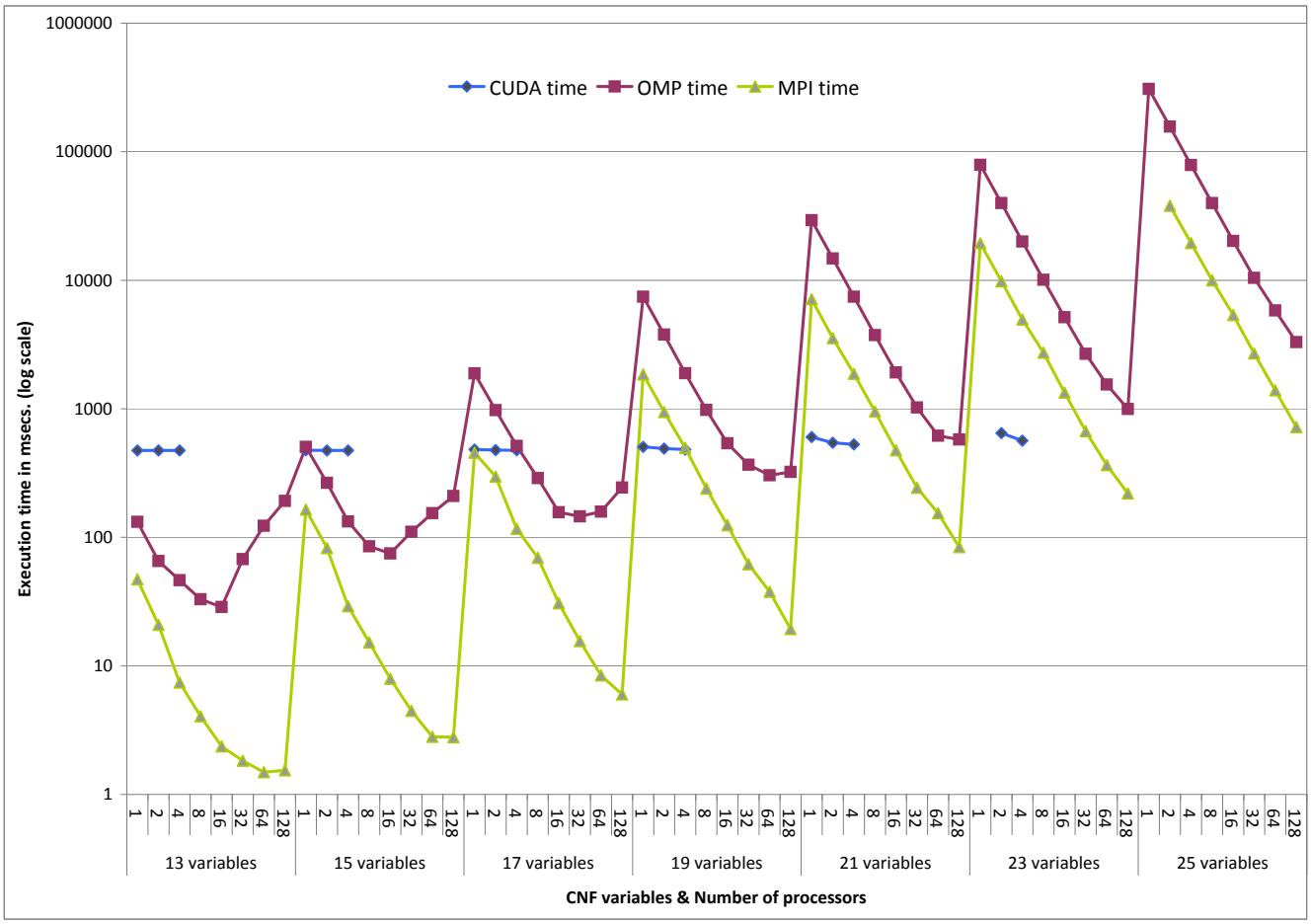


Figure 17: Execution time for the three different programming models and architectures: CUDA on GPUs, OpenMP on a shared memory system and MPI on a distributed memory platform.

7. REFERENCES

[1] Message Passing Interface (MPI). http://www.mcs.anl.gov/mpi.

[2] The OpenMP Specification. http://www.openmp.org.

[3] S. Alonso, L. Fernández, F. Arroyo and J. Gil. A circuit implementing massive parallelism in transition P systems. *International Journal Information Technologies and Knowledge*, 2(1):35–42, 2008.

[4] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. M. del Amor, I. Pérez-Hurtado and M. J. Pérez-Jiménez. Simulating a P system based efficient solution to SAT by using GPUs. *Journal of Logic and Algebraic Programming*, 79(6):317–325, 2010.

[5] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. M. del Amor, I. Pérez-Hurtado and M. J. Pérez-Jiménez. Simulation of P systems with active membranes on CUDA. *Briefings in Bioinformatics*, 11(3):313–322, 2010.

[6] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[7] D. Díaz, C. Graciani, M. A. Gutiérrez-Naranjo, I. Pérez-Hurtado and M. J. Pérez-Jiménez. Software for p systems. In *Gh.Paun, G.Rozenberg, A.Salomaa, editors*, The Oxford Handbook of Membrane Computing, pages 437–454. Oxford Univ. Press, 2009.

[8] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez and A. Riscos-Núñez. An overview of p-lingua 2.0. *Lecture Notes in Computer Science*, 5957:264–288, 2010.

[9] V. Nguyen, D. Kearney and G. Gioiosa. An extensible, maintainable and elegant approach to hardware source code generation in reconfig-p. *J. Logic and Algebraic Programming*, 79(6):383–396, 2010.

[10] NVIDIA. *CUDA Programming Guide 2.0*. 2008.

[11] G. Paun. Membrane computing. An introduction. *Springer-Verlag*, pages 9–419, 2002.

[12] G. Paun, T. Centre and C. Science. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 1998.

[13] M. J. Pérez-Jiménez, Á. Romero-Jiménez and F. Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *J. Natural Computing*, 2(3):265–285, 2003.

[14] M. Qasem. WinSAT website: (http://users.ecs.soton.ac.uk/mqq06r/winsat).

GPU-Accelerated Genetic Algorithms

Rajvi Shah
International Institute of Information Technology
Hyderabad, India
rajvi.shah@research.iiit.ac.in

P.J.Narayanan
International Institute of Information Technology
Hyderabad, India
pjn@iiit.ac.in

Kishore Kothapalli
International Institute of Information Technology
Hyderabad, India
kkishore@iiit.ac.in

ABSTRACT

Genetic algorithms are effective in solving many optimization tasks. However, the long execution time associated with it prevents its use in many domains. In this paper, we propose a new approach for parallel implementation of genetic algorithm on graphics processing units (GPUs) using CUDA programming model. We exploit the parallelism within a chromosome in addition to the parallelism across multiple chromosomes. The use of one thread per chromosome by previous efforts does not utilize the GPU resources effectively. Our approach uses multiple threads per chromosome, thereby exploiting the massively multithreaded GPU more effectively. This results in good utilization of GPU resources even at small population sizes while maintaining impressive speed up for large population sizes. Our approach is modeled after the GAlib library and is adaptable to a variety of problems. We obtain a speedup of over 1500 over the CPU on problems involving a million chromosomes. Problems of such magnitude are not ordinarily attempted due to the prohibitive computation times.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Parallel Programming*; I.2.8 [**ARTIFICIAL INTELLIGENCE**]: Problem Solving, Control Methods, and Search—*Heuristic Methods*

General Terms

Algorithms,Performance

Keywords

GAs,GPU,Genetic Algorithm,CUDA,Parallel GAs

1. INTRODUCTION

Genetic Algorithms (GAs) are a set of evolutionary algorithms, powerful and effective in solving search and optimization tasks. This class of algorithms is inspired by the process of biological evolution. Similar to the process of evolution, genetic algorithms employ natural selection, crossover, mutation and survival of fittest to find the fittest solution in a search space represented by a population of chromosomes, where each chromosome represents one possible solution to the optimization problem (Holland [4]).

A typical genetic algorithm starts with selecting random points in search space, representing them as chromosomes and building an initial population. This initial population is evaluated using a fitness function to suggest how fit a chromosome is to represent the solution. A fitness-based or uniform selection is carried out to select parent chromosomes to undergo crossover and produce offsprings, which usually with a very low mutation probability gets mutated. Hence, main components of a genetic algorithm are chromosome representation, selection, crossover and mutation. A representation that encodes the solution of the problem in the best possible way is used. Crossover and Mutation operators are often limited by the representation being used. Many methods exist for the process of selection as well, a method is chosen based on the convergence and diversity needs.

The user needs to tune various parameters and experiment with genetic operations and selection methods to achieve desired results using a genetic algorithm. In such a scenario, a library-like utility provides users great flexibility and ease of experimentation, speeding up the process of actual problem solving. Many public libraries exist for genetic algorithms providing a unified and optimized approach to achieve desired results. GALib (Wall [16]) is one such widely accepted library which enables the users to represent and solve their problems using genetic algorithms in a simple and effective way with enough flexibility. The long execution times associated with Genetic Algorithms constraints its application in many domains, despite its popularity on many domains.

In this paper, we present a generic framework for Genetic Algorithms accelerated by the modern Graphics Processing Units (GPUs), inspired by GALib. Such a framework not only provides a platform for fast execution but encourages experiments in new domains and with novel approaches involving huge population sizes which was limited due to impractical execution times. The key distinction of the approach is the effort to go beyond chromosome level parallelism whenever possible and utilize the massively multithreaded model of GPUs to its fullest.

 ISBN 978-84-693-6141-2

Our approach is implemented using Nvidia's CUDA programming model (NVIDIA [7]), but with enough isolation from the user program so that users need not be proficient in CUDA programming. We implement the original genetic algorithms and achieve a speed up of about 1500 on large problems using the massively multithreaded model of the GPU as exposed by CUDA.

2. RELATED WORK

Genetic Algorithms have been well explored and used in many domains for a long time. Attempts have been made in recent times to accelerate their performance using GPUs. Yu et al. [18] implemented a fine-grained parallel genetic algorithm (Tomassini and Calcolo [14]) on the GPU using Cg shader mechanism. A hybrid genetic algorithm (HGA) was proposed and implemented on GPU using the graphics pipeline and shading languages by Wong and Wong [17].

These above approaches were implemented and tested on Nvidida's GeForce 6800GT GPUs using the graphics pipeline. As the GPUs became more powerful and popular, they have become fully programmable parallel processing units. With the availability of high-level programming languages such as CUDA (NVIDIA [7]) and OpenCL (Khronos OpenCL Working Group [6]), researchers now see GPUs as a high performance multi-core processors. This has established a trend for General-purpose computation on GPUs (GPGPU) (GPGPU [3]).

In a recent work, Pospíchal et al. [8] presented a mapping of the parallel-island model of GA (Cantu-Paz [1]) to the CUDA architecture. This approach was implemented and tested on Nvidia's high-end CUDA compatible GPUs, namely, the 8800 GTX and GTX 285. The mapping of population islands to blocks benefits tremendously by fast access to shared memory resources within a block accelerating the performance many times. The island model is further explored to solve 0-1 knapsack problem in Pospíchal et al. [9]. The islands model is especially well suited to the block structure of CUDA with limited shared memory. This approach doesn't extend well to the general GA framework, which is the focus of this work.

3. GPU AND CUDA ARCHITECTURE

We present an overview of the CUDA programming and hardware models in this section. Please see (NVIDIA [7]) for more details about CUDA programming. Figure 1 depicts the CUDA programming model, mapping a software CUDA *block* to a hardware CUDA multiprocessor. A number of blocks can be assigned to a multiprocessor and they are time-shared internally by the CUDA programming environment. Each multiprocessors consists of a series of processors which run the *threads* present inside a block in a time-shared fashion based on the *warp* size of the CUDA device. Each multiprocessor further contains a small shared memory, a set of 32-bit registers, texture, and constant memory caches common to all processors inside it. Processors in the multiprocessor executes the same instruction on different data at any time. This makes CUDA an SIMD model. Communication between multiprocessors is through the device global memory which is accessible to all processors within a multiprocessor. Synchronization between threads of a block are possible. Synchronization across blocks is possible only at kernel boundaries.

The CUDA API provides a set of library functions which can be coded as an extension of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi-core co-processor. The code executes as threads running in parallel in batches of warp size, time-shared on the CUDA processors. Each thread can use a number of private registers for its computation. A collection of threads (called a *block*) runs on a multiprocessor at a given time. Threads of each block have access to a small amount of common shared memory. Synchronization barriers are also available for all threads of a block. A group of blocks can be assigned to a single multiprocessor but their execution is time-shared. The available shared memory and registers are split equally amongst all blocks that timeshare a multiprocessor. An execution on a device generates a number of blocks, collectively known as a *grid* Figure 1.

Each thread executes a single instruction set called the *kernel*. Threads and blocks are given a unique ID that can be accessed within the thread during its execution. These can be used by a thread to perform the kernel task on its part of the data resulting in an SIMD execution. Algorithms may use multiple kernels, which share data through the global memory and synchronize their execution either at the end of each kernel or forcefully using barriers.

4. GPU ACCELERATED GA

Genetic algorithm execution is a parallel process. That is, there is no dependency across the chromosomes of a population for the process of fitness evaluation and genetic operations. Hence, the entire population can be operated in parallel within a generation. To exploit the parallelism at a greater level, we form groups of threads to handle a single chromosome, thus mapping the problem to a massively multithreaded model for which GPUs are best suited. Currently, we have implemented the generic genetic algorithm with uniform and roulette wheel selection strategies, one point crossover and flip mutation (Goldberg [2]). Figure 2 shows the overall flow of the genetic algorithm framework onto a GPU.

4.1 Data Organization

In past, efforts were made to effectively utilize the parallelism of chromosomes by employing one thread per chromosome to perform fitness evaluation as well as genetic operations (Pospíchal et al. [8], Robilliard et al. [10]). The key difference of our approach is that we use several threads to perform these operations on a single chromosome, resulting in a better utilization of GPU resources. This is realized in practice by organizing the data in GPU memory in such a way that genes of each chromosomes can be accessed efficiently in a coherent manner by multiple threads handling it. This section describes organization of thread and data, used by various CUDA kernels.

Population is laid out in main memory of GPU, as a two dimensional $N \times L$ matrix such that columns refer to chromosomes and rows corresponds to genes within chromosomes as shown in Figure 3, where N is population size and L is

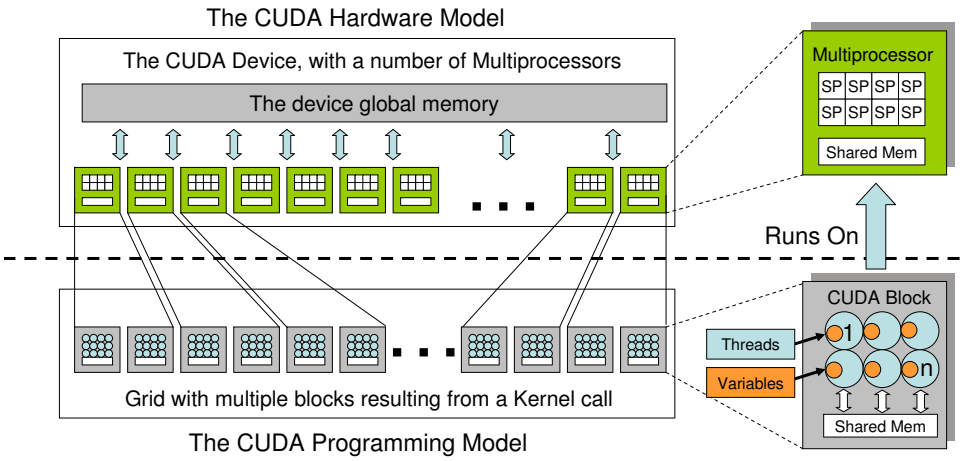


Figure 1: The CUDA hardware model (top) and programming model (bottom), showing the block to multiprocessor mapping.

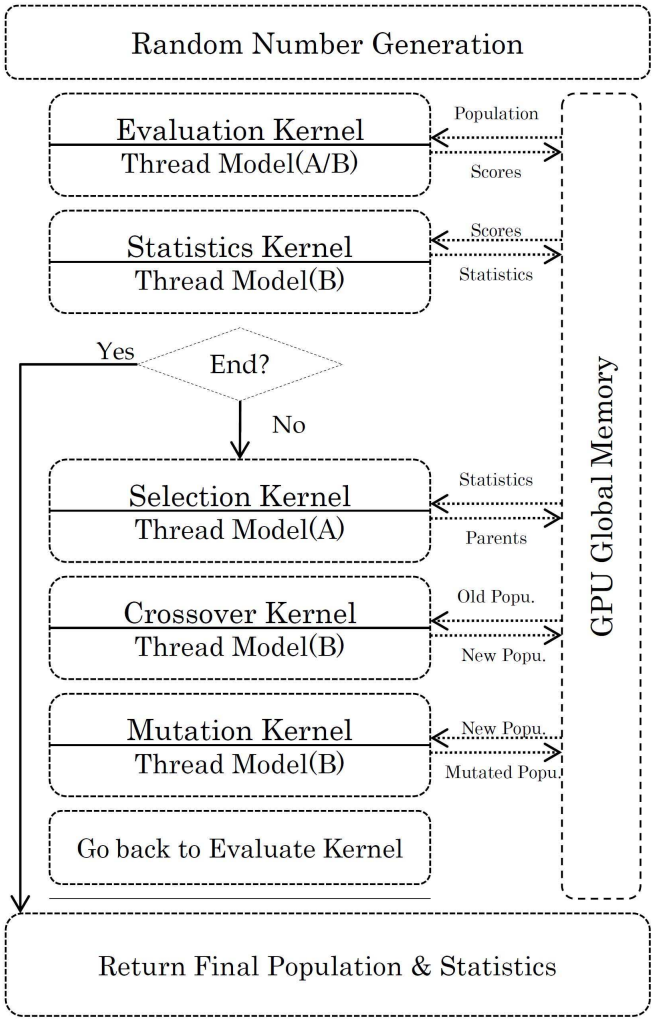


Figure 2: Program Execution and Memory Transfers

chromosome length. For a thread per chromosome model, threads in a block are arranged as a one dimensional array as shown in Figure 4 with one thread per chromosome. For a fully parallel approach, threads in a block are also arranged as a two dimensional matrix as shown in Figure 5. Kernel parameter *blockDim.x* is controlled by the number of threads per block (TPB), which is a CUDA block parameter. This layout leads to a one-to-one mapping between thread indices and genes. So, all genes of a chromosome can be accessed simultaneously.

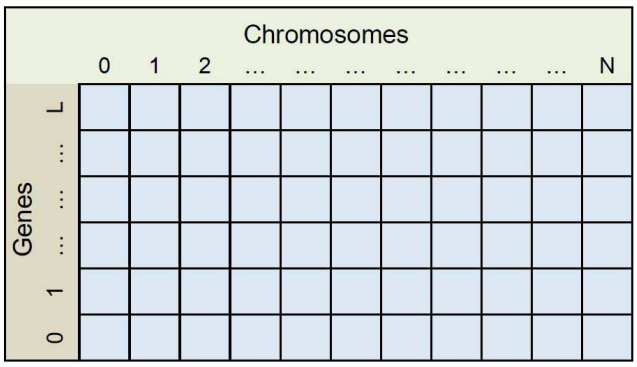


Figure 3: Population Matrix in memory

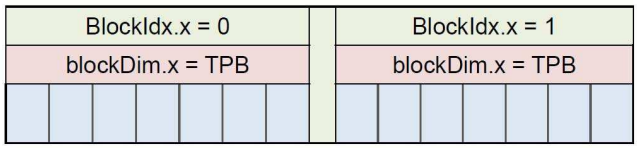


Figure 4: Thread Layout A

A detailed description of the execution flow depicted in Figure 2 and the mapping of the data layout shown in Figure 3 to a massively multithreaded model in each of the kernels is given in the subsequent subsections.

4.2 Fitness Evaluation Kernel

The process of fitness evaluation determines how fit each chromosome is to be the solution. Unlike other genetic op-

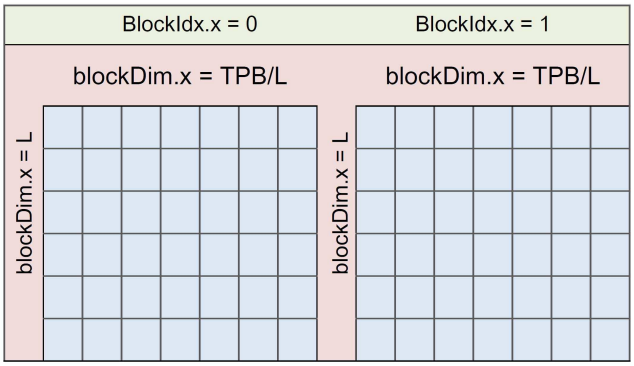


Figure 5: Thread Layout B

erators, fitness evaluation is a problem specific process and has to be provided by the user. In our framework, we provide a partially parallel and a fully parallel methods for the process of fitness evaluation.

The partially parallel method uses thread layout A as shown in Figure 4 with one thread per chromosome. In this method, user can access the chromosome as a 1D array and write an expression for fitness evaluation by accessing this array. User's C code fragment is used in fitness evaluation kernel by each of the threads to evaluate fitness of each individual. This is possible as CUDA is compatible to C. The calculated fitness scores are written back to the GPU global memory. As only chromosome level parallelism is exploited, this method may prove less efficient. But, it doesn't require a user to be familiar with CUDA architecture or programming. Hence, it makes the utility useful to a larger community at a small loss in performance.

The fully parallel method is provided for CUDA proficient users wherein the user can supply an evaluation function including a fitness evaluation kernel which may utilize the GPU resources in a more effective manner. This provides the user a way to achieve maximum performance.

Consider an example of 0-1 Knapsack problem. We are given a set of items with associated weights and costs. The aim is to pick items such that the total cost is maximum and total weight does not exceed knapsack capacity. A binary string is a convenient representation for chromosomes in this problem, where 1 indicates presence of an item and otherwise. Length of the chromosome is set to total number of items. In such a problem the fitness evaluation will involve finding cost sum and weight sum for all the chromosomes.

In partially parallel method, every thread will read one chromosome, its weight and cost, calculate total sum and total cost and write the score, providing parallelism across the chromosomes. Whereas in a fully parallel approach, we copy a block of cromosomes to shared memory. According to thread layout B (Figure 5), threads in each column read genes of corresponding chromosome, multiply it with cost and weight arrays and perform a log-sum as shown in Figure 6.

Fully parallel approach with careful utilization of shared resources can make the evaluation process much faster, espe-

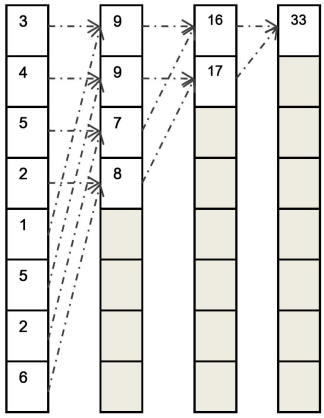


Figure 6: Parallel Sum

cially for problems involving intensive fitness calculation.

4.3 Statistics Kernel

After the fitness scores are calculated, population statistics need to be updated. Population statistics are used for the process of selection and to decide termination. The maximum, minimum and average and total fitness scores are calculated using standard parallel reduce algorithms (Jaja [5]). Best and worst chromosomes are recorded to ensure elitism, if selected by user. Also the selection probability for each of the chromosome is calculated.

Fitness scores may need to be sorted depending upon the selection method to be used. Sorting is not required if stochastic selection method is used. For probabilistic selection, like roulette wheel or rank selection, scores need to be sorted. A fast GPU based radix-sort, provided by CUDPP (CUDA Data Parallel Primitives) library is used for the same (Satish et al. [12]). Some method-specific statistics are also calculated, which is described later.

4.4 Selection Kernel

The execution of a Genetic Algorithm begins with the process of selection. In the process of selection, parent chromosomes are selected to go through the process of crossover to produce offspring. Selection Kernel will vary according to the selection method being used. Here, Uniform and Roulette Wheel selection kernels are described in detail. A uniform selection kernel is described in Pseudo-code 1.

Pseudo-code 1 Uniform Kernel

$N \leftarrow popSize$
$numThreads \leftarrow \frac{N}{2}$

{For all threads in parallel}

$i \leftarrow threadIdx$
$parent1(i) \leftarrow random(0, N-1)$
$parent2(i) \leftarrow random(0, N-1)$
$parent1(i+1) \leftarrow parent1(i)$
$parent2(i+1) \leftarrow parent2(i)$

Roulette wheel selection is more expensive than uniform selection. To simulate the roulette wheel, the population

is sorted based on the fitness score values (Satish et al. [12]). These score values are normalized to calculate selection probabilities. A sum-scan is performed on the normalized array (Sengupta et al. [13]). This new array is stored in global memory and used as a roulette wheel array (*rouletteArray*). These calculations are done in statistcs update stage, prior to execution of selection kernel. This selection kernel is described in Pseudo-code 2.

Pseudo-code 2 Roulette Wheel Kernel

```
GLOBAL : rouletteArray
N ← popSize
numThreads ← N/2

{For all threads in parallel}

i ← threadIdx
p1 ← random(0 − 1)
p2 ← random(0 − 1)

parent1(i) ← rotateWheel(p1, N)
parent2(i) ← rotateWheel(p1, N)
parent1(i + 1) ← parent1(i)
parent2(i + 1) ← parent2(i)
```

The rotateWheel function used in selection, performs a binary search on prefix-summed *rouletteArray* for the nearest smaller real number and returns parent index. This subroutine is described in Pseudo-code 3.

Pseudo-code 3 rotateWheel(n,N)

```
GLOBAL : rouletteArray
flag = 0
start = 0, end = N, middle = ⌈end/2⌉
while !flag do
  left ← rouletteArray[middle]
  right ← rouletteArray[middle + 1]
  if n >= left then
    if n < right then
      index ← middle
      flag = 1
    else
      start = middle
      middle = start + ⌈(end − start)/2⌉
    end if
  else
    end = middle
    middle = ⌈(end − start)/2⌉
  end if
end while
return(index)
```

Both, Uniform and Roulette Wheel Selection use thread layout A as shown in Figure 4.

4.5 Crossover Kernel

A pair of chromosomes selected in selection process undergoes the process of crossover to produce offsprings. The process of crossover is controlled by the crossover probability. Our implementation performs one-point crossover, but the same approach can be adapted to other crossover methods as well.

4.5.1 Crossover Preprocess

In our implementation, crossover points are calculated and stored prior to invoking actual crossover kernel. As the approach used for crossover uses multiple threads per chromosome, all the threads performing crossover between two chromosomes should know a common crossover point value. This prohibits generation of crossover points in crossover kernel itself due to a restrictive memory model and synchronization issues across the blocks in CUDA (NVIDIA [7]).

A kernel for selecting crossover points for one-point crossover is described by Pseudo-code 4. This kernel uses thread model A as shown in Figure 4.

Pseudo-code 4 Crossover Points Kernel

```
N ← popSize
L ← chromoLength
numThreads ← N/2

{For all threads in parallel}

i ← threadIdx
r1 ← random(0, 1)
if r1 ≥ probCross then
  crossPoint(i) ← random(0, L − 1)
  crossPoint(i + 1) ← crossPoint(i + 1)
else
  crossPoint(i) ← 0
  crossPoint(i + 1) ← 0
end if
```

In practice, crossover points are also selected along with parents in selection kernel as it uses the same thread layout.

4.5.2 One-point Cross-over

Instead of making one thread read two chromosomes, perform crossover and write the offspring chromosomes back, we make use of multiple threads to read a single chromosome. This approach results in a coalesced read and write of data speeding up the execution greatly.

As shown in the Figure 3, a chromosome occupies a column in the population matrix. Hence, the column index becomes the chromosome index. For the process of crossover we make use of total NL threads where N is the population size and L is the chromosome length. These threads are also laid out as a 2D matrix with N columns and L rows across the bloacks as shown in Figure 5. Now, instead of using one thread per crossover operation we use $2L$ threads, utilizing the massively multithreaded GPU model. Pseudo-code 5 describes a one-point crossover kernel using NL number of threads.

4.6 Mutation Kernel

In the process of genetic evolution, some chromosomes of the population mutate with a small mutation probability. Mutation is very crucial to bring genetic algorithm out of a local maxima or minima. The process of mutation is controlled by the mutation probability. We consider mutation probability as a probability for a gene to get mutated. For mutation kernel we again make use of matrix layout of Figure 3 for population and thread layout of Figure 5. Each

Pseudo-code 5 Crossover Kernel

$GLOBAL : Parent1, Parent2, crossPoint$
$N \leftarrow popSize$
$L \leftarrow chromoLength$
$numThreads \leftarrow N \times L$

{For all threads in parallel}

$C_{idx} \leftarrow threadIdx.x$
$R_{idx} \leftarrow threadIdx.y$

$p1 \leftarrow Parent1(C_{idx})$
$p2 \leftarrow Parent2(C_{idx})$
$xPoint \leftarrow crossPoint(C_{idx})$

if $R_{idx} \leq xPoint$ **then**
  $newPopulation(C_{idx}, R_{idx}) = oldPopulation(p1, R_{idx})$
else
  $newPopulation(C_{idx}, R_{idx}) = oldPopulation(p2, R_{idx})$
end if

thread now corresponds to a gene and decides whether or not to mutate the gene.

4.7 Random Numbers

Random numbers are extensively used throughout a genetic algorithm. CUDA does not provide any support for on the fly generation of a random number by a thread because of many synchronization issues associated. To solve this issue, an estimate of required random numbers is made. For example, a GA set up for a uniform selection with one-point crossover and flip mutation requires nearly $T = 2N+N+NL$ random numbers in one iteration, where N is the population size and L is length of the chromosome. Based on this estimate and memory limits imposed by hardware, a large pool of random numbers are generated and stored on GPU global memory before initiating the genetic algorithm. To speed up the process of generation and avoid transfer, we make use of rand routine provided by CUDPP, which uses MD5 algorithm for pseudo random number generation (Tzeng and Wei [15]). If high quality random numbers are needed, this is replaced by a CPU based random number generation followed by a copy to global memory.

5. PROGRAMMING INTERFACE

GALib (Wall [16]) is built around a few base classes, the main two being a Genome class and a Genetic Algorithm class. A user is allowed to tune a Genetic Algorithm according to the problem by setting various parameters through these classes.

Our framework is built around three main structures: GA Context, Genome Context and GAStatistics Context.

Out of these three, GA Context and Genome Context are mainly filled by user and contains various parameters for execution of genetic algorithm like population size, chromosome size, crossover and mutation probabilities, selection method, termination method etc. GAStatistics is mainly filled by the program along with execution of genetic algorithm. It holds fitness scores and other population related statistics. Support functions are provided to fill these structures, print parameters and destroy the structures.

Other than these three structures, user in his program needs to declare a void pointer to the population, and define and declare a user data structure which user might want to use for fitness evaluation. User also needs to supply fitness evaluation related code fragment as explained previously. A typical example user program 1 is listed below.

```
int main()
{
  void *population;
  UDATA udata;

  GAContext ga;
  GNMContext genome;
  GAStats stats;

  GASetParameters(&ga,&genome,&stats);
  GAPrintParameters(&ga,&genome,&stats);

  gaEvolvePopulation(&population,&ga,&genome
      ,&stats,&udata);
  PrintSolution(population,&genome,&stats);

  GAdestroyContexts(&ga,&genome,&stats);
  return 0;
}

__device__ float FitnessFunc(BIN1D *g,
    GNMContext genome,UDATA *udata)
{
        // Code to find fitness of a genome
        return score;
}
```

Program 1: An Example User Program

6. RESULTS AND DISCUSSION

We use a quarter of Nvidia's Tesla S1070 GPU to test our implementation. Tesla is a massively parallel platform with 30 multiprocessors each having 8 cores. We compare the performance of our GPU implementation with the serial implementation provided by GAlib (Wall [16]), running on an Intel Core2 Duo E7500(2.93 GHz) CPU. The direct comparison of running time (or speedup) of the CPU and the GPU is not entirely meaningful to evaluate their merits and demerits. The timing statistics is provided to provide the readers a sense of what they can expect with respect to a standard package.

We chose a standard 0-1 knapsack problem (Sahni [11]) to measure the performance speedup. We performed various experiments by changing chromosome length, population size, number of generations and selection methods. In all the experiments, the GPU accelerated approach showed significant speed up against serial implementation with comparable results. The quality of results is not degraded, as the basic algorithm was not modified but only paralleled.

A side-by-side performance comparison is given in Table 1 and Table 2, showing the GPU and CPU execution times for various population sizes for uniform selection and roulette wheel selection methods respectively. Chromosome length and number of generations were fixed at 50 and 100 respec-

tively. All the timings are averaged over 5 trials. The numbers clearly indicate that problems of huge magnitude can be solved in seconds with a GPU accelerated approach.

N	GPU	Std.Dev	CPU	Std.Dev	Speed Up
100	0.025	0.000006	0.127	0.006148	5.04
1000	0.031	0.000472	1.364	0.002490	43.61
10000	0.153	0.000485	19.799	0.270023	129.40
100000	1.561	0.001172	342.814	4.714645	219.06
1000000	3.662	0.001435	4803.381	214.557712	1311.36

Table 1: Timing Comparison for Uniform Selection, Time for 100 generations is given in seconds

N	GPU	Std.Dev	CPU	Std.Dev	Speed Up
100	0.046	0.000051	0.141	0.003834	3.01
1000	0.053	0.000041	1.629	0.017473	30.38
10000	0.209	0.000859	21.609	0.624903	103.19
100000	1.724	0.001149	492.927	9.326317	286.04
1000000	4.727	0.000327	7233.716	176.666921	1530.14

Table 2: Timing Comparison for Roulette Wheel Selection, Time for 100 generations is given in seconds

Table 3 shows the average execution time of the GPU-based approach for various chromosome lengths and population sizes for 100 generations, with a plot of the same in Figure 7. It is apparant from Figure 7 that the run-time growth is sublinear as the product NL increases.

	Population Size (N)				
L	100	1000	10000	100000	1000000
16	0.022	0.026	0.073	0.625	6.472
32	0.024	0.030	0.111	1.164	2.671
64	0.026	0.035	0.202	2.105	4.124
128	0.031	0.052	0.443	4.523	11.592
256	0.041	0.109	1.137	11.092	39.792
512	0.062	0.265	2.443	10.492	93.301

Table 3: Run-time in seconds for varying parameters for 100 generations

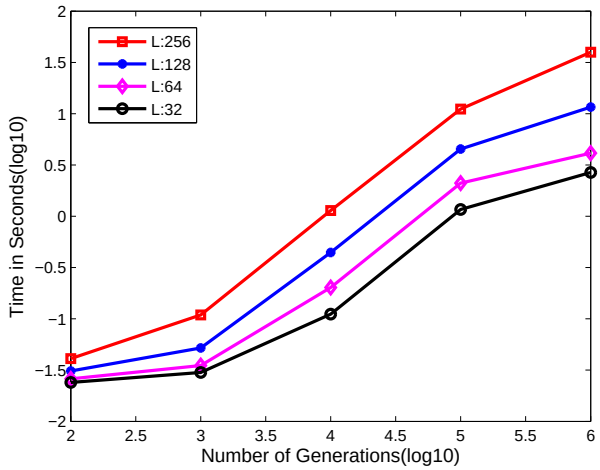


Figure 7: Run-time growth with N and L

Run-time growth of our approach with increasing number of iterations is linear as can be seen from Figure 8.

We also tried a numerical optimization problem using our implementation for its effectiveness. Our system found the minima of Rosenbrock's function effectively and fast.

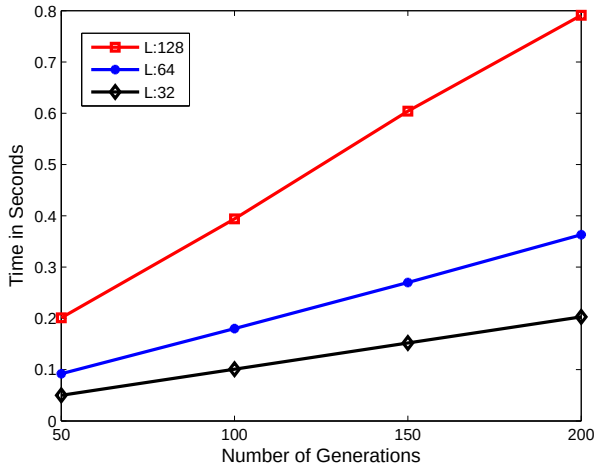


Figure 8: Run-time growth with number of generations

Direct performance comparison of our approach with other GPU based approaches is not meaningful. We achieve much higher speed up than that achieved by (Wong and Wong [17], Yu et al. [18]), but this comparison is not justified as they use a relatively old and less powerful hardware. Pospíchal et al. [8] use comparable hardware and demonstrates great speedup but using a parallel-island model of GA which can benifit greatly by shared resources.

7. CONCLUSION

In this paper, we demonstrate an approach to accelerate a simple genetic algorithm using the GPUs by exploiting gene level parallelism. We provide a mapping of various GA kernels to massively multithreaded model of GPUs using the CUDA programming model. Our GA framework is built around three basic structures to make the implementation extensible and flexible. Current implementation discusses a simple genetic algorithm with 1D chromosome and two different selection methods. The proposed framework can be extended to a GPU accelerated genetic algorithms library by incorporating more and more features. With speedup achieved over a factor of 1000 and a programmable library-like interface, GPU accelerated GA can find applications in many new domains.

8. ACKNOWLEDGEMENT

We thank Nvidia for providing equipment support.

References

[1] E. Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[2] D. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, 1989.

[3] GPGPU. General purpose computation on Graphics Processing Units. URL `http://www.gpgpu.org`.

[4] J. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press, Cambridge Mass., 1st MIT press ed. edition, 1992.

[5] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[6] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[7] NVIDIA. *NVIDIA CUDA Programming Guide Version 3.0*. NVIDIA Corporation, 2010.

[8] P. Pospíchal, J. JaroŽ, and J. Schwarz. Parallel Genetic Algorithm on the CUDA Architecture. In *Applications of Evolutionary Computation*, LNCS 6024, pages 442–451. Springer Verlag, 2010.

[9] P. Pospíchal, J. Schwarz, and J. JaroŽ. Parallel Genetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU. In *16th International Conference on Soft Computing MENDEL 2010*, pages 64–70. Brno University of Technology, 2010.

[10] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Population parallel gp on the g80 gpu. In *EuroGP'08: Proceedings of the 11th European conference on Genetic programming*, pages 98–109, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] S. Sahni. Approximate Algorithms for the 0/1 Knapsack Problem. *J. ACM*, 22(1):115–124, 1975.

[12] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[13] S. Sengupta, M. Harris, and M. Garland. M.: Efficient parallel scan algorithms for GPUs. NVIDIA. Nvidia technical report, NVIDIA Corporation, 2008.

[14] M. Tomassini and C. S. D. Calcolo. A Survey of Genetic Algorithms, 1995.

[15] S. Tzeng and L.-Y. Wei. Parallel white noise generation on a GPU via cryptographic hash. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 79–87, New York, NY, USA, 2008. ACM.

[16] M. Wall. GAlib, A C++ Library of Genetic Algorithm Components. http://lancet.mit.edu/ga/, 2008.

[17] M. Wong and T. Wong. Parallel Hybrid Genetic Algorithms on Consumer-Level Graphics Hardware. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2973–2980, 2006.

[18] Q. Yu, C. Chen, and Z. Pan. Parallel genetic algorithms on programmable graphics hardware. In *Advances in Natural Computation, First International Conference, ICNC 2005, Proceedings, Part III*, volume 3612, pages 1051–1059. Springer, August 27-29 2005.

Hybridizing Memetic Algorithms and Particle Filters for Visual Tracking on GPU

Raúl Cabido
Universidad Rey Juan Carlos
c/ Tulipán s/n, 28933
Móstoles (Madrid), Spain
raul.cabido@urjc.es

Antonio S. Montemayor
Universidad Rey Juan Carlos
c/ Tulipán s/n, 28933
Móstoles (Madrid), Spain
antonio.sanz@urjc.es

Juan J. Pantrigo
Universidad Rey Juan Carlos
c/ Tulipán s/n, 28933
Móstoles (Madrid), Spain
juanjose.pantrigo@urjc.es

ABSTRACT

Visual tracking is oriented at estimating the state of one or multiple moving objects in a video sequence. This is one of the first tasks in processing video systems which try to describe human behavior in different contexts, such as video-surveillance, sport technique analysis, etc. This work presents an object tracking system which properly hybridizes particle filters and memetic algorithms on a GPU architecture to produce a more reliable and efficient tracking algorithm. The system has been tested on synthetic and real image sequences with the aim of describing their performance under different conditions. Experimental results demonstrate that the MAPF algorithm accurately tracks objects in the scene with a precision and accuracy better than standard PF. In addiction, GPU implementation allow us to keep computational load and precision in proper balance.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*; I.3.m [**Computer Graphics**]: Miscellaneous; I.4.8 [**Image Processing and Computer Vision**]: Scene Analysis—*Tracking*

General Terms

Algorithms

Keywords

Memetic Algorithms, Particle Filtering, Hybrid Methods, Visual Tracking, Graphics Processing Units

1. INTRODUCTION

Video-based visual tracking is a complex task, consisting of the estimation of the position of a set of targets (people, vehicles, etc.) moving in the scene [14]. Most of the potential applications of this task (such as video surveillance, human-computer interaction, etc.) require from robust and efficient tracking algorithms. This is a relevant problem in Computer Vision and it has been focused using different methodologies. Several approaches have been proposed to tackle the problem of visual tracking [27] [28] [19]. Some of the most popular approaches are based on algorithms derived from the particle filter (PF) framework [23] proposed in [8].

This work presents an algorithm called memetic algorithm particle filter (MAPF) which hybridizes particle filters [8] and memetic algorithms [18][17]. This algorithm has been carefully chosen to be embedded in a parallel platform, such as a graphics processing unit, because of the parallel-friendly nature of the particle filter method [16][3].

This paper also presents experimental evidences of the system performance in relation to quality and computational load. The main contributions of this work can be summarized as follows:

1. Development of a reliable tracking system that is able to track targets in 2D in synthetic and real sequences.
2. Adaptation of a hybrid particle filter algorithm called memetic algorithm particle filter (MAPF) to a parallel and scalable platform.

The rest of the paper is organized as follows. Section 2 focuses on the description of the main algorithmic techniques and topics related to this work. Section 3 resumes the proposed approach and Section 4 illustrates the details of the MAPF application in visual tracking. Section 5 is devoted to describe the adaptation and implementation details of the MAPF to a GPU architecture. Section 6 shows and discusses the experimental results obtained. Finally, Section 7 summarizes the conclusions.

2. BACKGROUND AND RELATED WORK

In this section we provide information related to the main topics of this work: particle filters and different hybrid methods, memetic algorithms and GPU computation.

2.1 Sequential Estimation and Particle filters

Many interesting problems in science and engineering require estimation of the state of a system that changes over time using a sequence of noisy measurements made on the system [2]. Tracking problems, which consist of the estimation of the position of one or multiple targets moving in

 ISBN 978-84-693-6141-2

a scenario along time [14], are important examples of sequential estimation problems. The state-space modelling of these systems focuses on the state vector, which contains all relevant information required to describe the system under investigation. In tracking problems, for example, this information describes kinematic characteristics of the target as position, orientation, velocity, etc.

A particle filter (PF) is based on a large population of discrete representations (called particles) of the probability density function (*pdf*) which describes the evolution of a given system [4]. To achieve this goal, PF combines adaptation and prediction strategies. Isard and Blake adapted this algorithm to be applied to visual tracking in the middle 90's [11] and later, they proposed the CONDENSATION (*CONditional DENsity propagaTION*) algorithm [12]. This proposal makes possible the contour tracking in video sequences and, nowadays, constitutes the basis of most tracking algorithms based on probabilistic principles.

Particle Filters (PF) are algorithms in which theoretical distributions in the state-space are approximated by simulated random measures (also called particles) [4]. The state-space model consists of two processes: (i) an observation process $p(Z_{1:t}|X_t)$ where X_t denotes the system state vector and Z_t is the observation vector at time t, and (ii) a transition process $p(X_t|X_{t-1})$. Assuming that observations $\{Z_0, Z_1, \ldots, Z_t\}$ are sequentially measured in time, the goal is the estimation of the new system state at each time step. In the framework of Sequential Bayesian Modeling, the posterior *pdf* is estimated in two stages:

(a) Evaluation: the posterior *pdf* $p(X_t|Z_{1:t})$ is computed using the observation vector $Z_{1:t}$:

$$p(X_t|Z_{1:t}) = \frac{p(Z_t|X_t)p(X_t|Z_{1:t-1})}{p(Z_t|Z_{1:t-1})} \quad (1)$$

(b) Prediction: the posterior *pdf* $p(X_t|Z_{1:t-1})$ is propagated at time step t using the Chapman-Kolmogorov equation:

$$p(X_t|Z_{1:t-1}) = \int p(X_t|X_{t-1})p(X_{t-1}|Z_{1:t-1})dX_{t-1} \quad (2)$$

The aim of the PF algorithm is to recursively estimate the posterior *pdf* $p(X_t|Z_{1:t})$. This *pdf* is represented by a set of weighted particles $\{(\mathbf{x}_t^0, \pi_t^0), \ldots, (\mathbf{x}_t^N, \pi_t^N)\}$, where the weights $\pi_t^i \propto p(Z_{1:t}|X_t = \mathbf{x}_t^i)$ are normalized. The state of the system S_t can be estimated by, for example, the following expression:

$$S_t = \sum_{n=1}^{N} \pi_t^n x_t^n \quad (3)$$

Figure 1 represents a pseudocode for the PF algorithm. From an algorithmic point of view, the PF algorithm directs the temporal evolution of a particle set. Particles in PF evolve according to the system model and they are selected or eliminated with a probability which depends on their weight, determined by the *pdf* [4]. In visual tracking problems this *pdf* represents the probability of a target being in a given position in the image. As a consequence, the utility of the particle filter algorithm for visual tracking problems lies in the description of the temporal evolution of the system state.

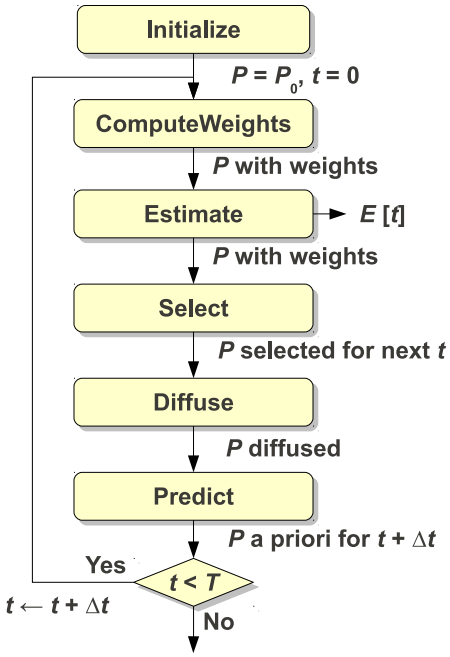


Figure 1: Flow diagram for the particle filter algorithm.

2.2 Hybrid particle filters

Unfortunately, the approach proposed by Isard and Blake is not effective in high dimensional estimation problems, such as multi-dimensional visual tracking (MVT) problems. In this category falls articulated and multiple object tracking. In the CONDENSATION approach, the number of required particles grows with the size of the state-space, as demonstrated in [15]. In fact, conventional PF algorithm does not scale well in problems with high dimensional state-spaces [25].

To address this difficulty, several optimized PF algorithms have been proposed. Most of them use different strategies to improve the performance of conventional PF algorithm. Partitioned Sampling (PS) [15] is a statistical approach to tackle hierarchical search problems. PS consists of dividing the state space into two or more partitions, and sequentially applying the dynamic model for each partition followed by a weighted resampling stage. Deutscher [6][7] developed an algorithm referred to as Annealed Particle Filter (APF) for tracking people. This is one of the first PF-based algorithms successfully applied to a multi-dimensional visual tracking problem. Furthermore, APF was the first algorithm which combined ideas from PF and metaheuristics (optimization strategies). Nowadays, there are several algorithms based on the hybridization of PF and Genetic Algorithms [5][7]. These algorithms include combination stages into the PF scheme, but no improvement stage is performed. Other methods have also been proposed, based on the use of Kalman filters [22], a combination of probabilistic and

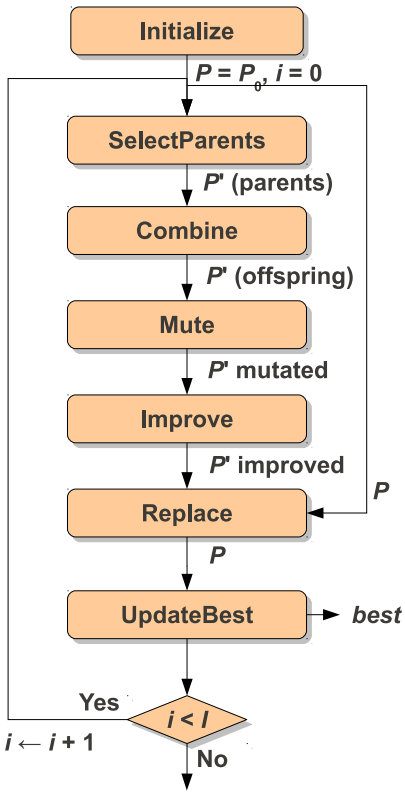


Figure 2: Flow diagram for the memetic algorithm.

evolutionary methods [24], etc. For example, a related approach combining a standard particle filter with a populational metaheuristic is presented in [26]. This paper introduces a coarse-to-fine algorithm for tracking licence plates in a sequence of video frames. The condensation algorithm is used for a coarse localization of the interest objects. At each iteration of this algorithm, additional iterations of Differential Evolution (DE) algorithm are embedded to optimize the license plate boundaries detection.

2.3 Memetic algorithms

The term memetic algorithm (MA) refers to family of metaheuristics that have as central theme the hybridization of different algorithmic approaches for a given problem [18][17]. This method has been successfully applied to a variety of optimization problems [10][18]. MA allows us to exploit all available knowledge about the problem under study [18]. That is what makes MA different from other evolutionary methods. This philosophy is illustrated with the term '*meme*' introduced in [**?**], which denotes an analogy with the '*gene*' in the context of cultural evolution. The ideas ('*memes*') are propagated from brain to brain via the cultural processes in a way similar to how the gene pool ('*genes*') is propagated from body to body via reproduction processes. This characterization of a '*meme*' suggest that in cultural evolution processes, information is not simply transmitted unaltered between individuals, but it is processed and enhanced by the communicating parts. In algorithmic terms, this enhancement is accomplished in MAs by incorporating heuristics, approximation algorithms, local search techniques, specialized recombination operators, truncated exact methods, etc. [18].

Therefore, the key idea of the MAs is the combination of different heuristics (tipically, individual improvement procedures with cooperation and competition processes) in a populational context. That is, MA maintains a population of solutions during the whole optimization process. These solutions are related to each other in a competition and cooperation context, which is organized in different generations. Each generation is a new update of the population, which is performed by recombining the features of some selected solutions and replacing some solutions with the new ones. The selection and replacement procedures are both competitive processes, while the combination stage is a cooperation process in which the selected solutions generate new ones by applying reproductive operators (typically, combination and mutation). Finally, the MA allows the application of local improvement procedures on some of the solutions. The improvement procedure can be used at different stages of the optimization process, for example: as a mutation operator, only at the end of the process, etc. Figure 2 shows a pseudocode for the MA.

2.4 Computer architecture and GPU computation

Computer architecture is evolving fast, its trend is moving forward from a single and fast execution unit with large memory space to several execution units with small local memory. Nowadays, multicore processors are common consumer systems. They are progressively replacing high energy-consumption desktop computers with great success and performance gains and the industry predicts that future computing systems will benefit from this scalable technology. This evolution is highly beneficial for data-parallel programs, where independent data processing takes place in the algorithms. An example of a future platform is the massively parallel graphics processing unit (GPU), with lots of execution units, small and fast memory for each processing core and high memory bandwidth. Using this platform for demonstration purposes, we can test algorithmic approaches that would scale well to new generations of desktop computers.

The term GPU (in contrast to the popular CPU) was coined by the manufacturer NVIDIA when its GeForce 256 card was released in 1999. The GPUs of the end of the 90's supported multitexturing capabilities and progressively increasing the number of available textures for improving realism of a 3D scenario. This hardware device was responsible for the float computation involved in rendering in a very parallel and efficient way, offloading the computational cost from the CPU. The next great revolution was the introduction of programmable stages in the graphics pipeline, which they could be programmed through graphics abstractions. Later on the unified architecture enabled the introduction of general purpose computation abstraction layers. Nvidia CUDA [20] is the hardware/software architecture that offers Nvidia GPUs as general purpose computation devices exposing their parallel processing nature to non-graphics-specialized developers. The importance of the CUDA program is the algorithm design, which has to be suitable for a parallel perspective.

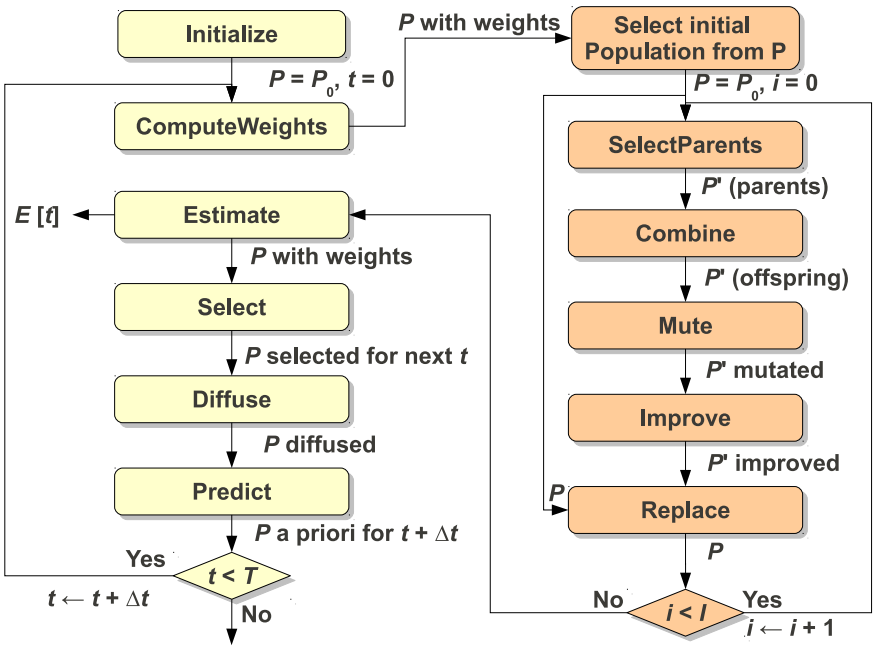


Figure 3: Flow diagram for the MAPF.

Taking into account this consideration we can adapt a particle filtering method to the GPU way of computation because of its usually large population of independent solutions, very prone to a parallel evaluation.

Shading programming was responsible of a young field, previously called GPGPU, as for general purpose computation using GPUs [9]. With the introduction of the unified architecture in 2006 and the Nvidia CUDA high level abstraction interfaces, GPUs became scalable and more easily programmed from the numeric software developer point of view, without the need of specialized graphics terminology. The hardware side of Nvidia CUDA is an array of streaming multiprocessors, while its software side is an extension of the C programming language that exposes Nvidia GPUs as parallel co-processors. It includes directives for GPU memory management and ways for invoking (launching) GPU programs (kernel functions) from a usual C application in order to process a (typically) large portion of data. It is based on the concept of grid of data blocks as a basic container, and thousands of light-threads are launched exploiting parallelism and each one executing a copy of the kernel function on each individual data. The typical CPU-GPU program would transfer data from the host memory (CPU side) to the device memory (GPU side). Then the device program, or collection of kernel functions, would operate on the data stored in the device memory and later we would send back the results to the host device. More information about programming using C for CUDA can be found on [13].

3. PROPOSED METHOD: MEMETIC ALGORITHM PARTICLE FILTER

MAPF is the result of hybridizing the memetic algorithm (MA) as optimization procedure and the particle filter (PF) as prediction procedure in two different stages:

- The PF stage focuses on the temporal evolution of a representative set of solutions. In this stage, a solution set called *SupportSet* of size N is propagated in time and updated to obtain a new set in every time step. The aim when using PF is to track multiple hypotheses and use the knowledge about the system dynamics for future prediction. Then, the key aspect for using PF is the prediction capabilities of the method to describe the temporal evolution of the system state.
- In the MA stage, solutions are combined and improved to obtain new and better ones using the strategies of the Memetic Algorithm metaheuristic. This stage is a refinement step embedded in the PF framework. PF algorithms are not suited to solve optimization problems, in contrast to MA algorithms. This is the main reason for using a metaheuristic as a refinement method in each time step.

Solutions in both MA and PF stages are codified as the same structure (see Section 4.1 for further details). Figure 3 shows a diagram for the MAPF algorithm. The proposed method follows the typical stages of a PF, adding a refinement stage after the particle weight computation, previous to the estimation. This refinement stage is based on a set of procedures extracted from MA general schema: selection, combination, mutation and improvement. When the refinement stage is finished, solutions are projected into the next time step using PF prediction strategies.

MAPF addresses its search towards regions of the solution space in which finding new better solutions is highly proba-

ble. MA stage performs a rational search beyond the simple stochastic procedure used by PF. On the other hand, PF stages increase the performance of general optimization algorithms in dynamic problems by improving the quality of the diverse initial solution set. MA and PF are related in such a way that when MA improves its results, PF performance also improves, and vice versa. PF makes parameter tuning possible, thereby adjusting the quality and the diversity of the particle set, used as diverse solution set by MA. On the other hand, MA improves the quality of the particle set, allowing a better estimation of the maximum of the *pdf*.

4. MAPF FOR VISUAL TRACKING PROBLEMS

This section illustrates the adaptation of the MAPF to be applied to a general multiple object tracking problem. A single object tracking problem is a particular case of the one depicted in this section.

4.1 Structure of a solution

The MAPF manages a population of solutions where each solution contains the set of required variables describing the system-state and its weight. The goal is to estimate the position of the objects in the scene. Therefore, the proposed state-space model for O object tracking is a $2 \times O$ dimensional space. The structure which stores a solution is a state vector $\mathbf{s}^t = [(x_1^t, y_1^t), \ldots, (x_O^t, y_O^t)]$, where (x_o^t, y_o^t) represents the position of the object o geometrical center in the global image frame at time t. The state $\mathbf{s}^t$ corresponds to a particle with an associated weight π^t. Besides, many other time-dependent objects features (such as their size, orientation, etc.) could be added to the solution structure.

4.2 Measurement Model

This subtask receives both a video frame at time t and the global scene static background image as input and returns the corresponding background subtraction image, binarized by thresholding. In mathematical terms, the resulting measurement image I_M^t at time t can be computed as:

$$I_M^t(l,w) = (|I^t(l,w) - I_B(l,w)| \geq th) \tag{4}$$

where $I^t(l,w)$ represents at frame t the intensity value of the pixel of coordinates l, w ($l \in [1, L]$ and $w \in [1, W]$, where L and W are the image length and width, respectively), I_B is the background image and th is a predefined threshold. In this work, we consider that the image regions belonging to the tracked targets are formed by white labeled pixels as a result of applying a fixed thresholding after a background subtraction. Depending on the application context, any other object detection method (i.e. colour segmentation, motion analysis, etc.) may also be suitable for this task.

4.3 Weighting Function

The weight π^t at time t assigned to each state described by a particle $\mathbf{s}^t$ in the PF stage is computed using the measurement image I_M^t as the sum of the white pixels belonging to the bounding boxes associated with each object:

$$\pi^t = \sum_{o=1}^{O} \pi_o^t \tag{5}$$

where π_o^t is a weight associated with each object o in the solution and O is the total number of targets. This is computed as follows:

$$\pi_o^t = \sum_{l=x_o^t-(Lx_o/2)}^{x_o^t+(Lx_o/2)} \left(\sum_{w=y_o^t-(Ly_o/2)}^{y_o^t+(Ly_o/2)} I_M^t(l,w) \right) \tag{6}$$

where (Lx_o, Ly_o) is the length and width of a bounding box fitting the object o. These parameters are pre-established according to the expected dimensions of the targets. The higher the number of labelled pixels contained in the bounding boxes associated with each object, the higher the likeliness of the particle is.

4.4 PF and MA selection stage

The selection stage is intended to improve the quality of the set, letting the best particles survive and replacing the worst ones by better estimators, their average, or even with the best one provided by the previous stage. We have implemented the PF selection simulating a roulette wheel selection procedure. In this way, we generate for each particle a uniformly distributed random value in the range $[0, \pi^{max}]$ where π^{max} is the maximum value of the weights in the particle set at each time step.

This random value is used as a threshold to evaluate whether its corresponding particle survives or is replaced based on its weight. Eventually, this procedure will replace most low quality particles as, in average, their weight values would not be very high. However, there should be some probability for low quality particles to survive because they can provide diversity to the set.

4.5 PF Diffusion and MA Mutation methods

The PF diffusion and the MA mutation methods are used to keep the needed diversity in the particle set. They are absolutely equivalent in our implementation and basically consist in a random perturbation of the spatial coordinates of a given particle:

$$\begin{cases} x'^t = x^t + F \\ y'^t = y^t + F \end{cases} \tag{7}$$

where x, y and x', y' denote the spatial variables before and after the perturbation, respectively, and F is a random uniform variable in a given range $[min_F, max_F]$. This is a reasonable implementation of the mutation method, taking into account that the solution is codified as a real valued vector.

4.6 PF System Model

The system model describes the temporal update rule for the system state [29]. The tracked object state consists of a given number of spatial coordinates and their corresponding velocities, deriving in a first-order system model.

$$\begin{cases} x^{t+\delta t} = x^t + \dot{x}^t \delta t + F \\ y^{t+\delta t} = y^t + \dot{y}^t \delta t + F \\ \dot{x}^{t+\delta t} = \dot{x}^t + G \\ \dot{y}^{t+\delta t} = \dot{y}^t + G \end{cases} \tag{8}$$

where x, y denote the spatial variables, $\dot{x}, \dot{y}$ are the first derivatives of x, y with respect to t, δt is the time step

and F, G are two excitation forces modeled by random uniform variables in a given range $[min_F, max_F]$ and $[min_G, max_G]$, respectively, which allow changes in the object state (position and velocity). The values of $[min_F, max_F]$ and $[min_G, max_G]$ depend on the expected changes in the position and velocity of the tracked object (Usually, $min_F = -max_F$ and $min_G = -max_G$).

4.7 MA Combination Method

This step is one of the most relevant of the MA for the optimization of the estimation achieved by the previous PF stage. Let $\mathbf{s}^a = [(x_1^a, y_1^a), \ldots, (x_O^a, y_O^a)]$ and $\mathbf{s}^b = [(x_1^b, y_1^b), \ldots, (x_O^b, y_O^b)]$ be any two solutions at time t[1], the combination method provides a solution $\mathbf{s}^c = [(x_1^c, y_1^c), \ldots, (x_O^c, y_O^c)]$ as follows:

$$\begin{cases} x_o^c = \dfrac{\pi_o^a x_o^a + \pi_o^b x_o^b}{\pi_o^a + \pi_o^b} \\ y_o^c = \dfrac{\pi_o^a y_o^a + \pi_o^b y_o^b}{\pi_o^a + \pi_o^b} \end{cases} \quad (9)$$

where π_o represents the weight of object o, $\forall o \in [1, O(t)]$. In other words, $\mathbf{s}^c$ is the result of the linear combination of the solutions $\mathbf{s}^a$ and $\mathbf{s}^b$, taking into account the weight contribution of each object in the scene at time t. Therefore, the better the object estimations are, the higher the contribution to the new solution is.

4.8 MA Improvement Method

Given an initial solution $\mathbf{s}^t = [(x_1^t, y_1^t), \ldots, (x_{O(t)}^t, y_{O(t)}^t)]$ and a neighborhood defined by a set of movements $N_\mathbf{s} = [\Delta x_1, \Delta y_1, \ldots, \Delta x_M, \Delta y_M]$, the improvement procedure generates new solutions starting from $\mathbf{s}^t$ and performing the unit movements in $N_\mathbf{s}$. For example, a local search iteration for the object o involves the following solutions:

$$\begin{cases} (x_o^t + \Delta x_o, y_o^t) \\ (x_o^t - \Delta x_o, y_o^t) \\ (x_o^t, y_o^t + \Delta y_o) \\ (x_o^t, y_o^t - \Delta y_o) \\ (x_o^t + \Delta x_o, y_o^t + \Delta y_o) \\ (x_o^t + \Delta x_o, y_o^t - \Delta y_o) \\ (x_o^t - \Delta x_o, y_o^t + \Delta y_o) \\ (x_o^t - \Delta x_o, y_o^t - \Delta y_o) \end{cases} \quad (10)$$

This procedure has to be repeated for each object in the solution. We follow a first-improvement strategy, which means that the process is initialized every time a better solution is found. The process ends when no improved solutions are found in the neighborhood considered.

5. IMPLEMENTATION DETAILS: MAPF ON GPU

In order to efficiently adapt the MAPF to the GPU framework for a visual tracking problem we have to ensure that we can access to the video stream from the GPU. We upload the RGB video frame as a graphics texture, we map it to a `cudaArray` and we fetch from the texture to compute the measurement model. It can be as straightforward as a map operation if the measurement model is a pixel operation like filtering, thresholding, background subtraction, etc.

[1]For the sake of simplicity, superscript t is omitted

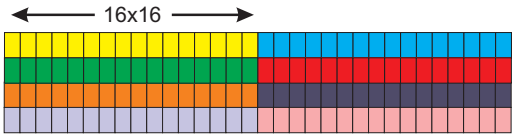


Figure 4: Scheme of the ROI encoding inside subarrays of size 16×16.

The result is attached as a pixel buffer object with the aim to visualize it if needed with OpenGL interoperability. Then we can proceed with the particle filter stages.

As stated in subsection 2.4 we need to process data on device memory using a number of kernel functions. We refer with the prefix `d_` those data stored in device memory.

The initialization kernel reads the states of the particles population (encoded in `d_states`) and creates a new buffer (`d_rois`) in device memory containing N regions of interest (ROIs) associated to the number of particles (`num_particles`) of the population. If possible, it is a good idea to encode the ROIs in subarray sizes power of 16×16 in order to ensure a good data layout and minimize thread ramification in the next kernels (see Figure 4). After this stage we would usually get a collection of binary regions in a large buffer where we can compute the likelihood of each solution contained in the particle set (particle evaluation). This evaluation is the most computationally demanding stage of the particle filter, as it needs to evaluate many pixel locations and perform certain operations on them. However, it can be very efficient for the GPU because of the high bandwidth to on-board memory, faster than system memory. We can use the CUDPP `scan` and `compact` operations to compute the particle weights by counting the number of labeled pixels inside every ROI.

At this point, we can embed the memetic algorithm to enhance the population. We take a number of solutions, typically 16-64 (`num_solutions`), randomly from the particle population, so `num_solutions`<`num_particles`. These solutions are formed by combining 2∗`num_solutions` parents from the particles states. These new solutions are evaluated as the previous implementation stages, reading from them and creating and evaluating new ROIs. Every new solution is included in the particle set replacing `num_solutions` particles in the particle set beginning from a random position in the `d_states` vector. This is done in a fast device memory copy operation instead of replacing random states inducing to random accesses. However, only the best solution is candidate for a local search (LS) phase.

The LS looks for better states (positions in our problem) starting from the best one of the previous stage and moving systematically through neighborhoods. The different neighborhoods are offline encoded in a vector `d_moves` and we use it to evaluate new ROIs as previously described. The best one is the final estimation of the MAPF. Note that the GPU parallel evaluation offers a *best improvement* approach instead of a *first improvement* typical serial evaluation scheme.

Table 1: Obtained framerates (measured in frames per second, fps) of the performance of the MAPF using 64, 256 and 512 particles in the PF stage and 16, 32 and 64 solutions in the MA phase using the CPU and GPU platforms.

N_{part}	N_{sols}	GPU	CPU	Speedup
64	16	166.0	57.26	2.90
64	32	164.0	56.79	2.89
64	64	162.0	55.09	2.94
256	16	132.0	37.70	3.50
256	32	130.5	37.60	3.47
256	64	130.0	37.20	3.49
512	16	104.0	25.27	4.12
512	32	103.5	25.20	4.11
512	64	102.5	25.18	4.07

The resampling stage rejects low quality states from the final particle set. In order to do so, a random number between 0 and the maximum weight is created for each particle. This number is used as a threshold. If a state has lower weight than its threshold it is replaced by the best estimator. Overall, this stage is a parallel resampling approach that improves the particle set quality letting the best particles survive, but also low quality ones providing diversity to the population.

A final diffusion stage would add some noise to the particle states, in order to spread repeated states from the previous stage.

6. EXPERIMENTAL RESULTS

This section is devoted to describe the results obtained by the tracking system. The experiments were performed on a 2.4GHz Intel®Quad Core with 4GB RAM using a Nvidia Geforce 8600GT GPU with 256 MB onboard and Nvidia CUDA 3.1 with drivers v257.21. We have also used OpenCV (Open Source Computer Vision) version 2.0 as a wrapper library to import video sequences to the C for CUDA application.

First of all we compare the benefit of the GPU platform measuring different configurations of the MAPF method on CPU and GPU. The CPU version was coded in C without SIMD optimizations or multicore programming, but was quite carefully optimized in the inner loops. Moreover, the strategy of the local search of the MA stage was a *first improvement* on CPU while on the GPU platform was a *best improvement*, that forces to evaluate an entire neighborhood in each iteration. More information about the CPU implementation can be found in [21]. Attending to the parameters taken into consideration in the configuration of a MAPF we can test it with different number of particles in the population of the PF and different number of solutions in the MA combination stage. Table 1 resumes this comparison showing the performance in frames processed per second (fps) of both platforms using a video resolution of 320×240 pixels and a ROI size of 32×32 pixels. In order to give a fair comparison, all the performance results on GPU include memory transfers from system memory to GPU onboard memory. We can observe how the GPU speedup increases in comparison to the CPU platform when the number of particles increases too. The number of solutions in the MA phase does not affect too much to the results in both platforms.

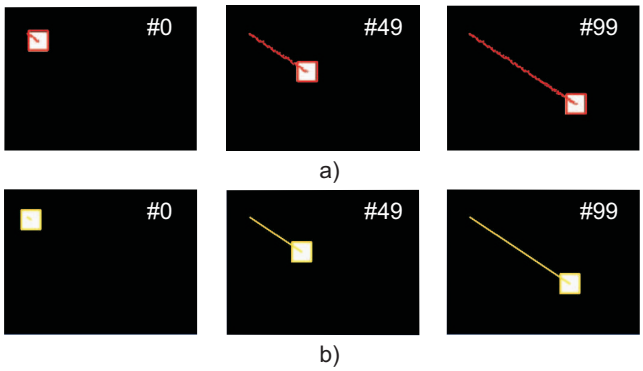


Figure 5: Tracking performance during a synthetic sequence of a 64x64 square moving from the top left corner to the bottom right one. a) PF using a population of 512 particles, b) proposed MAPF with a configuration of 64 particles, 32 solutions and 128 movements (approximately the same computational cost than the previous PF configuration). The MAPF performs better and no zig-zag trajectory is shown like in the PF tracking.

Figure 5 shows a synthetic sequence where a white square of size 32×32 pixels moves through the diagonal and a bounding box of the same size (32×32) is used for tracking. A particle filter with 256 particles is compared to a MAPF with only 64 particles, 32 solutions in the memetic algorithm and 128 movements in the local search phase. Both configurations are similar in terms of computational cost (as we will show later, 191 fps and 166 fps, respectively). The MAPF performs much better than the PF, showing a perfect tracking without any vibration. The PF does a good job although it shows some vibration on this perfect motion. To demonstrate this fact we compute a measure of the quality results provided by the MAPF algorithm, precision and accuracy of the estimates. Precision (also called reproducibility or repeatability) refers to the degree to which further measurements or calculations show the same or similar results. As a measure of precision in coordinate P_x, the average of the standard deviation of the estimations using series of independent measurements under the same experimental conditions was computed. In mathematical terms:

$$P_x = \frac{1}{F}\sum_{f=1}^{F}\sqrt{\frac{\sum_{m=1}^{M}(x_E^{f,m} - \hat{x}^f)^2}{M-1}} \tag{11}$$

where $x_E^{f,m}$ is the estimated value of the target's x-coordinate at frame f and experiment m, F is the total number of frames, M is the total number of experiments and $\hat{x}^f$ is the average value of target's x-coordinate over all experiments at frame f:

$$\hat{x}^f = \frac{1}{M}\sum_{m=1}^{M} x_E^{f,m} \tag{12}$$

Table 2: Quality results ($F = 240$ and $M = 10$) and framerates (in fps) of the PF and MAPF on GPU for video resolutions of 320×240 pixels.

Method	N_{part}	N_{sols}	P_x	P_y	A_x	A_y	fps
PF	256	-	0.88	0.94	0.92	1.07	191
PF	512	-	0.70	0.72	0.58	0.62	139
PF	768	-	0.64	0.63	0.47	0.47	109
MAPF	64	16	0.01	0.00	0.01	0.01	166
MAPF	64	32	0.00	0.00	0.00	0.00	164
MAPF	64	64	0.00	0.00	0.00	0.00	162

Accuracy is the degree of conformity of a measured or calculated quantity with respect to its true value. Therefore, the accuracy computation requires the knowledge of the real values of the variable. The accuracy A_x in the x coordinate was computed as follows:

$$A_x = \frac{1}{MF} \sum_{m=1}^{M} \sum_{f=1}^{F} (x_T^f - x_E^{f,m})^2 \quad (13)$$

where x_T^f and $x_E^{f,m}$ are the true and estimated values of the target's x-coordinate, respectively, F is the total number of frames and M is the total number of experiments. Similar equations can be applied to compute the precision and accuracy for the y-coordinate (P_y and A_y, respectively).

Table 2 shows the quality of the methods when applied to the tracking of a synthetic object of dimension 32×32 pixels using a bounding box of the same size as in the previous figure. Note that the PF method does not show values in the N_{sols} column as they are referred to the memetic algorithm stage. The MAPF method can easily outperform the PF and shows an almost perfect tracking as the problem is relatively simple for the local search performed in the MA phase. Frames per second (fps) are also shown in the last column and it can be observed how the performance of the PF with different configurations is about the same as the MAPF but quality results are a bit lower. Also, the performance of the MAPF remains constant as it can compute 16, 32 or 64 combinations in negligible time compared to the rest of the computation.

The proposed MAPF has been also tested with the public standard benchmark CAVIAR [1]. Figure 6 shows an example of the results obtained by the proposed system on the *Walk1* sequence of the database. A person walking from the bottom right corner to the upper left corner is tracked in real time (exhibiting performances of more than 100 fps) using a configuration of 64 particles in the PF stage and 32 solutions in the MA stage.

7. CONCLUSIONS

In this paper, we have presented an effective hybridization of a standard particle filter with a memetic algorithm for the problem of a single object visual tracking in real time. This new algorithm, called MAPF, has been also implemented using a stream programming model like Nvidia CUDA. The particle filter method acts as the sequential estimation filter in which a memetic algorithm refines its population in order to improve the estimations for the tracking problem. The friendly nature of the particle filter as well as the good adaptation of the memetic algorithm into the core of the PF has leaded to very good results in terms of overall performance and quality.

As future work, we propose the extension of the MAPF to a multidimensional tracking problem such as a multiple object tracking or an articulated object tracking in both, synthetic and real video sequences.

8. ACKNOWLEDGMENTS

This research has been partially supported by the Spanish projects TIN2008-06890-C02-02.

9. REFERENCES

[1] Caviar test case scenarios. http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1/.

[2] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for on-line nonlinear/non-gaussian bayesian tracking. *IEEE Trans. on Signal Processing*, 50(2):174–188, 2002.

[3] R. Cabido, A. S. Montemayor, J. J. Pantrigo, and B. R. Payne. Multiscale and local search methods for real time region tracking with particle filters: local search driven by adaptive scale estimation on gpus. *Mach. Vision Appl.*, 21(1):43–58, 2009.

[4] J. Carpenter, P. Clifford, and P. Fearnhead. Building robust simulation based filters for evolving data sets. Technical report, Dept. Statist., Univ. Oxford, Oxford, U.K, 1999.

[5] J. Cui and Z. Sun. Vision-based hand motion capture using genetic algorithm. In *Lecture Notes in Computer Science*, volume 3005, pages 289–300, 2004.

[6] J. Deutscher. Articulated body motion capture by annealed particle filtering. In *IEEE Conference on In Computer Vision and Pattern Recognition*, pages 126–133, 2000.

[7] J. Deutscher and I. Reid. Articulated body motion capture by stochastic search. *Int. J. Comput. Vision*, 61(2):185–205, 2005.

[8] N. J. Gordon, D. Salmond, and A. Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. *IEE Proceedings F Radar & Signal Processing*, 140(2):107–113, 1993.

[9] GPGPU: General-purpose computation using graphics hardware. http://www.gpgpu.org, 2010.

[10] W. Hart, N. Krasnogor, and J. Smith. *Recent Advances in Memetic Algorithms*. Springer-Verlag New York, Inc., 2005.

[11] M. Isard and A. Blake. Visual tracking by stochastic propagation of conditional density. In *4th European Conf. Computer Vision*, pages 343–356, 1996.

[12] M. Isard and A. Blake. Condensation - conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29:5–28, 1998.

[13] D. B. Kirk and W. mei W. Hwu. *Programming Massively Parallel Processors: A Hands On Approach*. Morgan Kaufmann, 2010.

[14] J. MacCormick. *Stochastic Algorithms for Visual Tracking: Probabilistic Modelling and Stochastic*

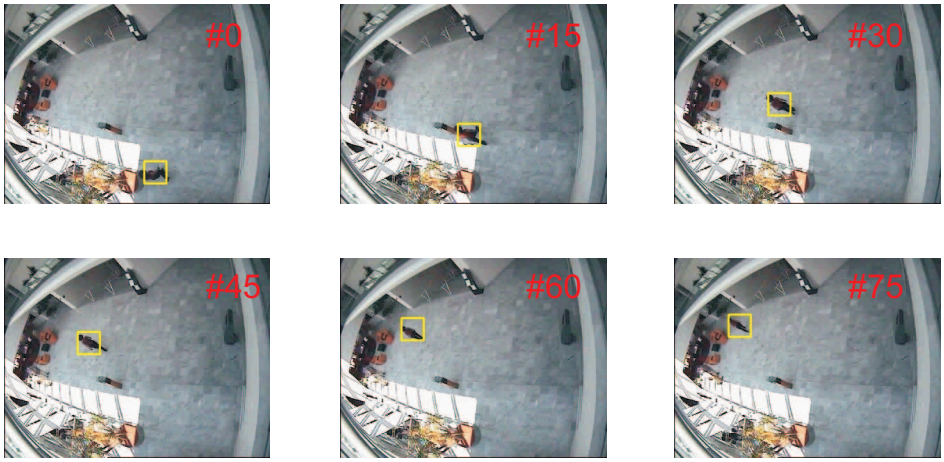


Figure 6: Tracking performance of the MAPF in a real sequence from the CAVIAR database.

Algorithms for Visual Localisation and Tracking. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[15] J. MacCormick and A. Blake. Partitioned sampling, articulated objects and interface-quality hand tracking. In *7th European Conference on Computer Vision*, volume 2, pages 3–19, 2000.

[16] A. S. Montemayor, J. J. Pantrigo, A. Sánchez, and F. Fernández. Particle Filter on GPUs for Real Time Tracking. In *ACM SIGGRAPH 2004 posters*, 2004.

[17] P. Moscato. *Memetic Algorithms: a short introduction*, pages 219–234. McGraw Hill, 1999.

[18] P. Moscato and C. Cotta. *A gentle introduction to Memetic Algorithms*, pages 105–144. Kluwer Academic Publishers, 2003.

[19] W. Ng, J. Li, S. Godsill, and J. Vermaak. Tracking variable number of targets using sequential monte carlo methods. In *IEEE Statistical Signal Processing Workshop*, pages 1286–1291, 2005.

[20] NVIDIA. CUDA Zone. http://www.nvidia.com/object/cuda_home.html, 2010.

[21] J. J. Pantrigo, J. Hernández, and A. Sánchez. Multiple and variable target visual tracking for video surveillance applications. *Pattern Recognition Letters*, 31(12).

[22] C. Rossi, M. Abderrahim, and J. C. Díaz. Tracking moving optima using kalman-based predictions. *Evol. Comput.*, 16(1):1–30, 2008.

[23] S. Särkkä, A. Vehtari, and J. Lampinen. Rao-blackwellized particle filter for multiple target tracking. *Information Fusion Journal*, 8:2–15, 2007.

[24] S. Shen, M. Tong, H. Deng, Y. Liu, X. Wu, K. Wakawbayashi, and H. Koike. Model based human motion tracking using probability evolutionary algorithm. *Pattern Recognition Letters*, 28:1877–1886, 2008.

[25] S. Thrun. Particle filters in robotics. In *17th Annual Conference on Uncertainty in AI (UAI)*, 2002.

[26] I. K. Yalçin and M. Gökmen. Integrating differential evolution and condensation algorithms for license plate tracking. In *ICPR '06: Proceedings of the 18th International Conference on Pattern Recognition*, pages 658–661, Washington, DC, USA, 2006. IEEE Computer Society.

[27] A. Yilmaz, O. Javed, and M. Shah. Object tracking: A survey. *ACM Comput. Surv.*, 38(4):13, 2006.

[28] L. Zhu, J. Zhou, and J. Song. Tracking multiple objects through occlusion with online sampling and position estimation. *Pattern Recognition*, 41(8):2447–2460, 2008.

[29] D. Zotkin, R. Duraiswami, and L. Davis. Joint audio-visual tracking using particle filters. *EURASIP Journal on Applied Signal Processing*, 11:1154–1164, 2002.

A Parallel Memetic Algorithm for Workload Distribution in Dynamic Multi-Agents Systems

David Millán Ruiz
Telefónica R&D
Emilio Vargas, 6
Madrid, Spain
+34-913372638
dmr@tid.es

J. Ignacio Hidalgo
Complutense University of Madrid
School of Informatics
Madrid, Spain
+34-913947537
hidalgo@dacya.ucm.es

ABSTRACT
This paper describes a parallel evolutionary approach to the problem of workload distribution in dynamic multi-agent systems based on blackboard architectures. Specifically, we focus on the multi-skill call centre use case. This type of call centres entails quick adaptations to a changing environment that only some greedy algorithms have been able to cope with. These greedy heuristics consist of a continuous re-planning, considering the current state of the system. As these decisions are greedily taken, the workload distribution may be poor for middle and/or long term planning due to incessant wrong movements. The use of parallel memetic algorithms, which are much more complex than classical, ad-hoc heuristics, can guide us towards more accurate and robust solutions. Now, the difficulty underlies in how to apply these techniques to uncertain, ever-changing environments. Specifically, in previous studies, we proposed a neural network to make accurate predictions and a single memetic algorithm as a heuristic optimisation mechanism, improving the results obtained by other well-known techniques of the call centre domain. In this study, we upgrade our approach by parallelising the memetic algorithm and carrying out a deeper analysis.

General Terms
Algorithms, Experimentation.

Keywords
Parallel Memetic Algorithms, Dynamic Multi-Agent Systems, Multi-Skill Call Centre.

1. INTRODUCTION
Over the last decade, a gradually-growing interest in parallel and distributed computing has arisen in computer science. This concern has guided research activities towards areas such as parallel and distributed programming, distributed information systems, and parallel and distributed hardware architectures. Truthfully, there exists a vast bibliography (e.g. see [1, 4, 17]) related to this issue, although there are still paths to explore.

Furthermore, we perceive a tendency to tackle increasingly complex problems and application domains which commonly involve the processing of uninterrupted, dynamic data flows. These demanding environments are usually hard to be efficiently maintained by conventional and sequential techniques. Nevertheless, parallel and distributed methods not only mitigate this drawback but also present several valuable characteristics such as robustness, traceability, problem simplification, adaptivity, scalability and speed-up.

Conversely, dynamics, synchronisation and behaviour appear as intricacies of parallel and distributed information systems because the representation of linear problems into sub-problems is not always feasible or simply straightforward.

Anyhow, parallel and distributed systems should somehow self-improve to attain high performance. In fact, nowadays, a wide range of studies on adaptive techniques in parallel and distributed information systems can be found [2, 8].

A classical, well-suited problem for studying dynamic systems is the workload distribution in Multi-Agent Systems (MAS) [21]. The term *intelligent agent* [10] describes an autonomous entity which is able to observe and interact with its environment in order to accomplish a given set of tasks [19]. When several agents interact, they may compile a multi-agent system. Characteristically, such agents have a partial point of view of the problem and thus need to cooperate with other agents. Furthermore, there may be no global control and consequently such systems are sometimes denoted as *swarm systems*. In these cases, data are decentralised and execution is asynchronous. Although there are numerous types of MAS, we will focus on those Dynamic Multi-Agent System (DMAS) encapsulated in blackboard architectures [9, 11]. In other words, we will work on DMAS with a common repository of knowledge.

Commonly, the basic variant of the problem of workload distribution in a DMAS requires the assignment of tasks to agents which have the required skills to handle them over time, fulfilling a predefined set of additional constraints and respecting the dependencies among individual tasks and differences in the execution skills of the agents.

This problem has diverse variants but, depending on the dynamism of the system, we can principally distinguish two main scenarios:

1. On the one hand, we can find short-term planning environments in which a continuous planning is needed due to the high dynamism of the system. These solutions attempt to distribute the workload among agents by applying “basic” ad-hoc heuristics, looking at the current system state (without predictions or predictions for a short term time-frame). This

feature can be effortlessly seen in workload allocation within a *dynamic multi-skill call centre* [3].

2. On the other hand, we can find long-term planning systems in which the list of tasks is predefined and known by all agents like in the classic *scheduling problem* [6]; or environments in which a single task type is assigned to each agent for a long period of time, similarly to the *job assignment problem* [7]. In other cases, agents are assigned to patterns of tasks, instead of specific tasks (such as in *pattern-based scheduling* [6]). Analogously, stable multi-skill call centres [3] can be also included in this group. These solutions consider stable behaviour over time, anchored in historical data and apply more complex algorithms to match agents and task types. However, when having a dynamic system, these approaches cannot be efficiently applied, since an adaptive method is required.

Our proposal is encapsulated in the first scenario (dynamic systems) as Figure 1 illustrates.

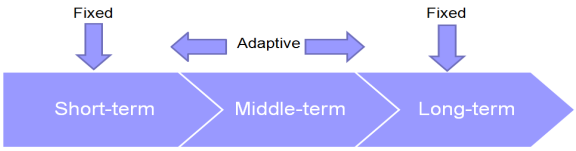


Figure 1. Adaptive time-frame.

Table 1 summarises some key features of the environments described in the preceding paragraphs, according to the time-frame considered.

Table 1. Comparison of the time-frame size considered for the workload distribution problem

Time frame	Complexity	Response time	Adaptability	Performance
Short	Low	Low	medium	medium
Middle	high	Medium	high	high
Long	medium	High	low	low

Specifically, we will focus on a specific use case which is the multi-skill call centre as it covers all the general characteristics of a typical DMAS.

In the study [14], we presented a hybrid system (consisting of a neural network to predict upcoming system states and a "basic" Memetic Algorithm (MA) to optimise the allocation of workload for those predicted system states) which improved the most representative techniques of the state of the art. The assumption behind the solution proposed in [14] can be summarised as: "*in highly dynamic systems, it is preferable to make strategic plannings for a middle-term time-frame (assuming certain noise as predictions are not 100% accurate) than to make poor plannings for a short-term time-frame repeatedly*".

Ad-hoc heuristics intelligently consider the current state of the system but, due to time constraints, they have to execute very simple operations to route each task to the corresponding agent (or sometimes to groups of agents) in near real time. However, if we could make predictions of future states for a middle-term time-frame with enough accuracy, we could apply more sophisticated techniques that would enable us to obtain accurate enough solutions for these stages. Obviously, we are missing some "snapshots" of intermediate states along the way but, if the "time jumps" are small enough, we can assume certain seasonality in the system. Therefore, system changes will be minimal and any optimal solution for that system state will be potentially appropriate for the whole interval. Depending on the dynamics of the system, time-frames must have greater or lesser extent. The size of the time-frame can be automatically determined by analysing the variability of the predictions provided by the neural network. To analyse that variability, we measure the mean absolute error made at *t-v* (latest time-frame), where *v* is the size of the prediction window expressed in terms of seconds.

The main contribution of this work lies in the extension and parallelisation of the approach presented in [14]. We also provide a deeper testing than the one done in [15] (different data sets, more executions and a deeper analysis the results).

The rest of this paper is organised as follows: Section 2 presents the problem definition adapted to the multi-skill call centre use case, highlighting the complexity of this application domain. Section 3 summarises how the problem was solved in [14, 15] and explains how that work has been extended for this paper. Section 4 provides an evaluation of the model and promotes an analysis of results. In this evaluation, we will compare our approach with other acknowledged techniques. Finally, Section 5 concludes by summarising our work and provides some guidelines for future work.

2. PROBLEM DEFINITION

In a common DMAS, there are n tasks or work items grouped in k types of tasks and m agents that may have up to l skills ($l \leq k$) to perform these works. In this manner, each agent can process different types of tasks and, given a type of task, it can be carried out by several agents that have that skill. The set of skills an agent has is frequently denoted as *profile*. These profiles can be truly heterogeneous as there are massive potential skills.

Although agents may have multiple skills, each agent can only process one operation at the same time. Furthermore, given an operation, it requires an unknown amount of time to be accomplished. Besides, each agent must orderly process each operation during an uninterrupted period of time; in other words, the task cannot be divided or postponed once it has already started.

As we have already mentioned, we will focus on the multi-skill call centre use case. Therefore, we will adapt the definition of a classical DMAS to the call centre domain.

A Call Centre (CC) [5] is a centralised office used for receiving and transmitting large volumes of telephone requests which may range from customer service to the selling of products and services. In a CC, the flow of calls is often divided into outbound and inbound traffic. Outgoing calls are handled by agents, primarily, with commercial pretensions. This type of calls is planned as agents know in advance which customers must be contacted every day. Conversely, incoming calls are those that go

from the client to the CC to contract a service, ask for information or report a problem. These unplanned calls are initially modelled and thus classified into manifold Call Groups (CGs) in relation to the nature of each call (complaints, V.I.P. clients, client loyalty, etc.). As soon as these CGs have been modelled, each call is assigned to a unique CG (there is no overlap among CGs).

A specific type of CC is the Multi-Skill Call Centre (MSCC). In an MSCC, there are *n* customer calls grouped in *k* types of calls and *m* agents that may have up to *l* skills ($l \leq k$). This implies that each agent can attend different types of calls and, given a type of call, it can be answered by several agents that have that skill.

Figure 2 illustrates the relationship among client calls, queues and agents. This figure describes an example for *9* client calls grouped in *4* CGs and *5* agents having different real skills.

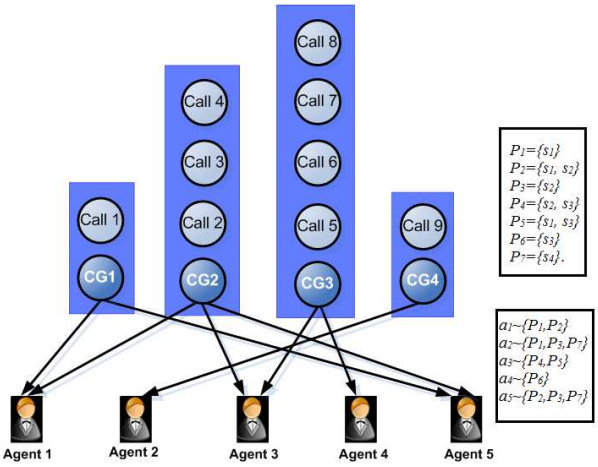


Figure 2. Multi-skill call centre configuration based on the potential skills of all agents.

More formally speaking, the following parameters can be found in an MSCC:

- a finite set of *n* customer calls $C = \{ c_1, c_2, \ldots, c_n \}$.
- a finite set of *k* CGs (call groups/types) $CG = \{ cg_1, cg_2, ..., cg_k \}$, where $k \leq n$ when every CG has, at least, one call queuing.
- a finite set of *m* agents $A = \{ a_1, a_2, ..., a_m \}$. Note that, usually, $m >> k$.
- a finite set of *k* agent-skills $S = \{ s_1, s_2, ..., s_k \}$ in which each agent-skill, s_i, represents the ability to handle the associated CG, cg_i, with the corresponding sub-index in CG: $s_1 \sim cg_1, s_2 \sim cg_2, ..., s_k \sim cg_k$.
- a finite set of *d* agent-skill profiles $P = \{ P_1, P_2, ..., P_d \}$ in which each agent-skill profile P_i can be any subset of $S = \{ s_1, s_2, ..., s_k \}$.
- a finite set of *n* operations (execution or processing of each customer call, c_i) $O = \{ o_1, o_2, ..., o_n \}$ in which each operation, o_i, has associated a mean processing time which depends on its CG: $\{\tau_1, \tau_2, ..., \tau_k\}$.

The solution to the problem of the workload distribution in MSCCs is defined as the right assignment for every agent a_i to the most suitable skill profile P_j from his/her real skill profiles for each *v* seconds, where *v* is the size of the time-frame considered.

In addition, the assignment $\langle a_i, P_j \rangle_t$ must satisfy all hard constraints and handle the soft ones given by the business units. To determine whether (or not) a given solution is suitable, we need to define a quality metric to evaluate the rightness of each feasible solution. There are very significant metrics to measure the quality of a CC such as the abandonment and service rates. These metrics somehow hinge on the (customer) *service level* [13] which is defined as the percentage of customer calls that have to queue shorter than a specified amount of time (*20* seconds in our case). Our work has been conducted by applying this metric.

Moreover, the solution must fulfil the following descriptions:

- on *O* define *R*, a binary relation which represents the precedence among operations. If $(o_1, o_2) \in R$ then o_1 has to be performed before o_2.
- each agent, a_i, has associated a finite non-null subset of *P*, containing his skills to handle different customer CGs (individual real skill-profile).
- the same profile P_i can be assigned to several agents. In other words, several agents may have some skills in common (or even all of them).
- every agent, a_i, may have several profiles assigned but only one can be performed at a given instant *t*, $\langle a_i, P_j \rangle_t$. In other words, an agent cannot process two (or more) incoming calls at the same instant.
- every solution must respect diverse (hard and soft) constraints given by business rules defined by business units or agents' regulations.

The complexity of this problem is huge because we are not only dealing with an NP-hard problem like the job assignment problem, but also considering high dynamism, massive incoming customer calls and large number of agents having multiple skills. Besides, since customer calls are not planned, this makes the call assignment a truly laborious task.

3. EXTENSION OF THE MODEL

This section summarises the methodology employed by our approach in [14] and how we have extended it in this study. In [14], we present an approach consisting of two main modules: a predictive module and a search module. The purpose of the present paper is to extend [14] by parallelising the MA and presenting the link between both modules as Figure 3 shows.

The search module and the forecast module need each other to properly distribute the workload among agents. However, there are other steps in between which link these two modules.

The first step consists in determining the right size of the time-frame by analysing the system variability. Once the right size for the time-frame has been established, we must forecast all variables of next system state at time *t+v*, ξ_{t+v} (*v* is the size of

the time-frame). These predictions are made by means of a forecast module which relies on an artificial neural network which is fully described in [18]. Given the predictions from the forecast module, the search module, implemented as a parallel steady-state MA, optimises the assignment among task types and agents.

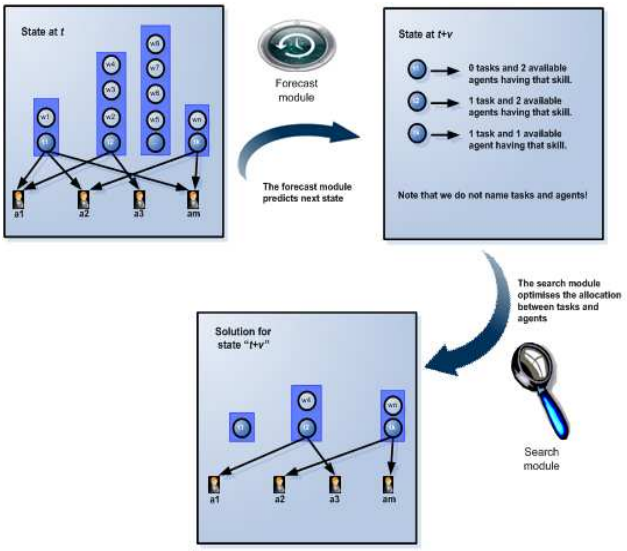


Figure 3. Overall process → forecast module + search module.

We propose an island topology and migration operators for individuals exchanging. We will consider a master island and several subordinate islands. Each island corresponds to a single MA. Each MA maintains a set (population) of abstract representations (chromosomes) of candidate solutions (phenotypes) to the problem described in Section 2. The population is partially randomly initialised. Then, its individuals are evaluated by applying a fitness function over them. From this population, some individuals are selected and, then, recombined (crossover). Subsequently, the offspring may suffer mutations in some genes. Afterwards, some of these individuals replace others from the population according to the replacement scheme. Every generation includes all previous actions. An LS mechanism is applied over a percentage of the population each *g* generations. All these steps are carried out in each island until a predefined time has been elapsed. Note that all the islands cooperate for a common goal, exchanging their best fitted individuals.

3.1 Encoding, Initialisation and Population

We will encode every solution as an array of integers whose indexes represent the available agents at a given instant and the array contents refer to the profile assigned to each agent. Figure 4 shows a fictitious example of encoding, related to Figure 2, for *9* customer calls (c_1-c_9) queued in *4* different CGs (cg_1-cg_4) depending on the nature of the calls, *5* agents (a_1-a_5) and *7* profiles (P_1-P_7), where $P_1=\{s_1\}$, $P_2=\{s_1, s_2\}$, $P_3=\{s_2\}$, $P_4=\{s_2, s_3\}$, $P_5=\{s_1, s_3\}$, $P_6=\{s_3\}$ and $P_7=\{s_4\}$. Now, suppose that the agents have the following potential skill profiles: a_1~$\{P_1,P_2\}$, a_2~$\{P_1,P_3,P_7\}$, a_3~$\{P_4,P_5\}$, a_4~$\{P_6\}$ and a_5~$\{P_2,P_3,P_7\}$. We have seen the potential profiles for every agent but only one profile can be assigned to each agent at a given instant *t*; therefore, a feasible solution would be Figure 4. Note that more than one agent can have assigned the same profile (e.g. a_1 and a_5).

Index (agents) →	1	2	3	4	5
Content (profiles) →	2	7	4	6	2

Figure 4. Example of encoding for an MSCC.

The population contains *20* different individuals encoded as hinted above. In our case, we propose to start from a randomly generated initial population, including the best solution found in the previous time-frame because the configuration of agents' profiles should not change too much over two successive time-frames (consecutive states).

3.2 Fitness Function

Now, we present the fitness function which is defined over the proposed encoding to measure the quality of a given solution. Our fitness function is inspired in the estimation of the *total service level* provided in [13] although we also consider the priority of each CG weighted as follows:

$$Total\_service\_level = \sum_{i=0}^{k} (Pr_i \times SL_i(\gamma_i, \alpha_i)) \times \mu \quad \textbf{(1)}$$

$\{sl : \Re \times [0,1] \times [0,1] \rightarrow [0,1]\}$ where *k* refers to the number of CGs, μ is a normalising factor $(1/\sum_{i=0}^{k} Pr_i)$, Pr_i is the priority of the CG_i whose service level is defined as $SL_i(\gamma_i, m_i) = 1 - P(Agents\_are\_busy) \times e^{-(\gamma_i - m_i)\frac{\tau_i}{\beta}}$ given that $P(Agents\_are\_busy) = \left[1 + \frac{\gamma_i - m_i}{m_i} \sum_{\zeta=0}^{\gamma_i - 1} \frac{(\gamma_i - 1)...(\zeta + 1)}{m_i^{\gamma_i - \zeta - 1}}\right]^{-1}$ where γ_i is the load of CG_i (number of incoming calls of CG_i by the mean processing time: $n_i \times \tau_i$), m_i is the number of agents of CG_i (based on the profiles assigned in the chromosome), τ_i is the number of agents of CG_i and β is the duration of the time-frame expressed in seconds.

Additionally, we handle some hard and soft constraints derived from the business rules given by our business units. In our case, these constraints are associated to tasks, agents, timing, actions or desired/undesired scenarios. Thus, the algorithm cannot violate hard constraints (e.g. we cannot change agents' profiles continuously due to certain laws and regulations); although we allow certain movements which may imply the violation of some soft constraints (e.g. we should not take agents from CGs in which the service level is below a given threshold). Undoubtedly, this type of movements is penalised according to the degree of non-accomplishment of these constraints and their relevance.

Therefore, the *fitness function* can be formalised as follows:

$$f = total\_service\_level - constraints\_penalisations \quad \textbf{(2)}$$

($f:[0,1]\times[0,1]\rightarrow[-1,1]$) where *constraints_penalisation* is the value obtained after applying our business rules (e.g. agents from CG-*i* should not move to CG-*j*).

Finally, we can speed-up the evaluations by introducing a *partial fitness function*. The first time, we need to employ (2) but the rest of the time; we just need to evaluate those groups affected by a mutation or, in the case of the LS, when generating a new neighbour. Hence, we only process the affected CGs in (1) and update their original values. With this information, we then recalculate (2).

3.3 Evolutionary Operators

In this section, we explain the final configuration of the evolutionary operators. This configuration is the following one:

- *Selection*: Since the population needs to be bred each successive generation, we have chosen a binary tournament selection.
- *Crossover*: The following step is to produce a new generation from selected individuals. We consider that children will inherit the common points in their parents and randomly receive the rest of genes from them.
- *Mutation*: This operator causes tiny changes in the genes of the chromosome to explicitly maintain diversity (actually there are much more mechanisms). We apply a perturbation over each gene of the chromosome with a probability of *0.03*. This perturbation corresponds to changes of profiles in some agents (e.g. agent a_2 who had assigned the *profile* P_1 has now associated the *profile* P_3 due to a mutation).
- *Replacement policy*: Finally, we decide which individuals are incorporated (or maybe reinserted) into the population. In this study, we consider elitism with a probability of *0.93* to replace the worst individuals of the population for next generation. And, with a probability of *0.07*, a worse individual may be captured. Note that our MA relies on a steady-state scheme.

The configuration proposed above has not been chosen ad-hoc. Instead, we have evaluated different configurations and selected the best one.

3.4 Local Search

LS is a Meta-Heuristic (MH) for solving optimisation problems. An LS algorithm starts out from a candidate solution and, thus, iteratively moves to a neighbour solution, generating the neighbourhood. To carry out this action, a neighbourhood relation must be defined on the search space. In our case, we state that two candidate solutions are neighbours if only one gene differs in both chromosomes. Note that we propose a simple LS due to the lack of time of our production environment (*300* seconds).

The following pseudo-code illustrates the LS algorithm which is applied to:

```
void Local_Search (Chromosome & candidate_solution)
  Chromosome best_solution = candidate_solution;
  Chromosome neighbour = candidate_solution;
  For (i=0; i<candidate_solution.size(); i++)
    Agent a = neighbour.getAgent(i);
    For (j=0; j<a.get_number_profiles(); j++)
      neighbour.change_profile(i,j); //profile j for agent i
      If (neighbour.fitness() > best_solution.fitness())
        best_solution = neighbour;
    neighbour = best_solution;
  candidate_solution = neighbour;
```

3.5 Memetic Algorithm

Once we have the evolutionary operators, we need to define the refinement mechanism and select a target subpopulation to refine. Another important issue is the refinement frequency.

- *Refinement algorithm*: The refinement mechanism is the basic LS described in Section 3.4.
- *Subpopulation for LS*: The LS is applied over the best *25%* of individuals.
- *LS frequency*: The LS is applied over the selected individuals each *10* generations.

3.6 Parallel Memetic Algorithm

The last step is to parallelise the MA. We will apply an island model (each island is an MA) with several subordinate islands connected to a master island as follows:

- *Topology*: We consider a star topology with *4* subordinate islands (as Figure 5 illustrates) which correspond to "simple" MAs. These islands are connected to a master island (another "simple" MA which coordinates and synchronises the rest of islands).
- *Migration*: Each subordinate island sends the *10%* of the best fitted individuals when the master island asynchronously demands these individuals to the rest of islands.
- *Replacement policy*: We will apply elitism so that the best fitted individuals from the subordinate islands will replace the less fitted individuals from the master island's population whether those individuals are better fitted.
- *Migration frequency*: Each *50* generations, the master island blocks the rest of islands to ask them for their best fitted individuals.

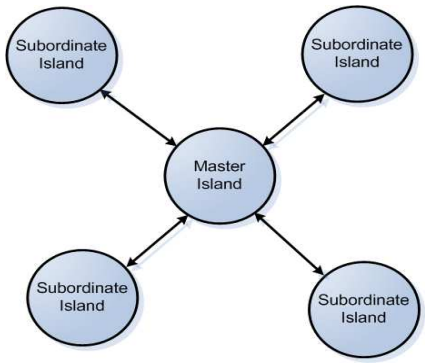


Figure 5. Star topology with *4* subordinate islands and a master island.

4. EVALUATION OF RESULTS

In this section, we analyse the results obtained by applying the configuration of the Parallel MA (PMA) proposed in this article. We assume that the forecast module is accurately feeding the search module (which is the case) with the upcoming system state for a properly determined prediction window (by analysing the dynamism of the system).

Afterwards, we will compare our PMA with the "simple" MA (SMA) proposed in [14] to verify the importance of this parallelisation. We will also analyse the results obtained by applying other well-known MHs such as Simulated Annealing (SA) [12], Iterated Local Search (ILS) [20] and Variable Neighbourhood Search (VNS) [16]. For this comparison, we will launch experiments for two different problem instances (with medium and high difficulty, respectively). These two instances are real data taken from our MSCC's production environment during two different days at the same hour (from *12:40* to *12:45*, *300* seconds): a one-day campaign and an ordinary day. The size of the time-frame to execute all the MHs is *300* seconds (*5* minutes) because we need to provide the system with a solution exactly each *300* seconds. We have selected this time interval because this hour (between *12:30* and *13:00*) is very representative as this is precisely the most critical hour of the day (highest arriving load of the day: *n/m*). Note that around *800* incoming calls (*n*) simultaneously arrive during a normal day in such a time interval, whereas up to *2450* simultaneous incoming calls may arrive during this interval during a commercial campaign. The number of agents (*m*), for each time interval, oscillates between *700* and *2100*, having *16* different skills for each agent on average (*minimum=1* and *maximum=108*), grouped in profiles of *7* skills on average. The total number of CGs considered for this study is *167*. Therefore, when the workload (*n/m*) is really high, finding the right assignment among agents and incoming calls becomes fundamental. In this way, we have run every MH under two double-core processors of a *Sun Fire E4900* server (one processor for the interfaces and data pre-processing, and the other one for each MH).

Once the magnitude of our MSCC has been presented, each MH is compared alongside the others. Table 2 summarises the results obtained by each MH in *50* executions, starting from *50* different randomly generated initial solutions.

In our comparative study, we present dissimilar MHs which cover diverse strategies. Theoretically, due to the local character of the basic LS, it is complicated to reach a high-quality solution because the algorithm usually gets stuck in a neighbourhood when a local minimum is found. This occurs because its search engine is always looking for better solutions which sometimes do not actually exist in the neighbourhood. For this reason, occasionally, it is more appropriate to allow deterioration movements in order to switch to other regions of the search space. This is precisely the smart policy of SA whose temperature (cooling simulation) allows for many oscillations (the probability of accepting a worse solution decreases over time) at the beginning of the process and only few fluctuations at the end (fewer chances to select a worse solution as the algorithm is supposed to be refining the solution at this point). Specifically, we have chosen Cauchy's criterion because the convergence is faster than Boltzmann's and we only have *300* seconds to run the complete process. Besides, this scheme avoids decreasing the distance between two solutions when the process converges (jumps in the neighbourhood). Therefore, the temperature must be high enough at the beginning to better explore the search space (its neighbourhood) and low enough at the end to intensify the search as well (exploitation of promising areas). The value for speed is, therefore, the stopping condition which must agree with the number of neighbours generated.

Table 2 gathers the results obtained by each MH in *50* different executions for two different problem instances with the purpose of providing a fair comparison. The first three columns are the best, worst and mean fitness values, respectively. Then, we have the standard deviation and the effectiveness (best fitted solution represents the *100%*).

We perceive from Table 2 that SA behaves worse than other MHs except for the easiest instance of the problem. This may occur because we are not plenty of time in our environment and the power of SA relies on a progressive cooling. If we cool off the temperature too fast, we are missing the effectiveness of accepting worse solutions in some cases. Instead, if we cool off the temperature too slowly, we may be accepting worse solutions systematically without converging. We have applied a trade-off between exploration and exploitation but the time seems to be limited to apply SA to our environment (perhaps, things might change when having more time).

Another option to increase the diversity in the solutions is to enlarge the environment, as VNS does. This philosophy consists of making a systematic change upon the environment when the LS is used, increasing the environment when the process becomes stagnated. In the VNS, the search is not restricted to only one environment as in the basic LS; instead, the neighbourhood changes as the algorithm progresses. Albeit we only consider three distinct neighbourhoods, the improvement of the VNS compared to basic LS is noteworthy. Consequently, the remarkable factor becomes the change in the number of neighbourhoods and their sizes as well as to consider how the algorithm reacts in response.

Table 2 also shows how VNS only slightly outperforms SA for the hardest instance of the problem.

Another strategy is to start from different initial solutions as ILS accomplishes. ILS generates a random initial solution and afterwards applies a basic LS. Subsequently, this solution is systematically mutated and thus refined. ILS obtains solutions which vaguely improve those given by SA and VNS for the hardest problem instance, although it performs worse for the simplest problem instance as Table 2 corroborates.

Another way to find a solution involves using methods based on populations, such as MAs. If the diversity of the solution is low, then the MA converges to the closest neighbour. Nevertheless, when the selective pressure is high, individuals may be alike or even identical. To speed-up the convergence, MAs apply an LS upon a set of chromosomes (candidate solutions) that are refined every certain number of generations. Incorporating a hybridisation mechanism to the GA is valuable as the algorithm is improved in all respects. This fact is pointed up in Table 2 as the MA not only outperforms all the strategies for both instances but also remains more unwavering (less differences among best, worst and mean fitness values).

Table 2. Results obtained by the MHs in *50* executions, starting from randomly generated initial solutions for two problem instances: medium and hard (larger number of incoming calls and high variability). Values refer to the fitness obtained by all MHs.

Algorithm	Best Solution		Worst Solution		Average		Standard Deviation		Effectiveness	
	Medium	Hard	Medium	Hard	Medium	Hard	Medium	Hard	Medium	Hard
PMA	0.834	0.818	0.823	0.783	0.829	0.809	0.003	0.002	100	100
SMA	0.796	0.758	0.785	0.751	0.796	0.754	0.001	0.001	96.01	93.20
ILS	0.768	0.728	0.755	0.722	0.763	0.725	0.002	0.003	92.03	89.61
VNS	0.790	0.727	0.766	0.723	0.775	0.724	0.005	0.001	93.48	89.49
SA	0.782	0.721	0.773	0.709	0.779	0.716	0.001	0.003	93.96	88.50

Finally, it is important to remark that differences among techniques are not huge after reaching a fitness of *0.8* since the complexity increases exponentially in our environment.

Therefore, minor improvements on the fitness value after that point are hard to obtain but very valuable to accomplish a fair workload distribution.

5. CONCLUSIONS AND FUTURE WORK

We have presented a parallel evolutionary approach to the problem of workload distribution in dynamic multi-agent systems based on blackboard architectures (common repository of knowledge). We have seen that these systems are extremely complex and involve quick adaptations to a varying environment that only high-speed greedy heuristics can deal with. These greedy heuristics consist in a permanent re-planning, considering the present system state. However, these quickly taken decisions are not appropriate for middle and/or long term planning due to the incessant erroneous movements.

However, we have demonstrated that the use of parallel memetic algorithms, which are more versatile than classical heuristics, can guide us towards more accurate solutions. With the intention of applying parallel memetic algorithms to such a dynamic environment, we have put forward a reformulation of the traditional problem of workforce distribution in dynamic multi-agent systems based on backboard architectures, which coalesces predictions of future system states with a precise search mechanism, by dynamically enlarging or diminishing the time-frame considered. We have claimed that the size of the time-frame depends upon the dynamism of the system (smaller when there is high dynamism and larger when there is low dynamism).

Our approach has been tested out on a real-world production environment from Telefónica which is one of the largest telephone operators around the world.

The present work has also illustrated how nearly optimal solutions each *v* seconds (size of the time-frame) outperforms continuous bad distributions when the right size of the time-frame is determined, and predictions and optimisations are correctly carried out. Particularly, we have put forward a neural network for predicting future system variables and a parallel memetic algorithm based on an island scheme to perform the assignment of incoming tasks to the right available agents.

We can conclude that PMAs not only outperform other MHs but also remain more unwavering as systematically provide better results.

As future work, we propose to do a similar study considering parallel MHs and different time-frame sizes.

6. REFERENCES

[1] Andrews, G.R.: *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison–Wesley, ISBN 0-201-35752-6, 2000.

[2] Asiki, A.; Tsoumakos, D. and Koziris, N.: *An Adaptive On-line System for Efficient Processing of Hierarchical Data*. Proceedings of the 18th International ACM Symposium on High Performance Distributed Computing (HPDC'09), Garching, Germany, 2009.

[3] Avramidis, A.N.; Chan, W.; Gendreau, M.; L'Ecuyer, P. and Pisacane, O.: *Optimizing daily agent scheduling in a multiskill CC*. European Journal of Operational Research (2009).

[4] Baeza-Yates, R. and Ribeiro-Neto, B.: *Modern Information Retrieval*. Acm Press Series, Addison Wesley, 1999.

[5] Bhulaii, S.; Koole, G. and Pot, A.: *Simple Methods for Shift Scheduling in Multiskill Call Centers*. M&SOM 10(3), 411–420, 2008.

[6] Brucker, P.: *Scheduling algorithms*. 2nd edn. Springer, Heidelberg, 1998.

[7] Chauvet, F.; Proth, J.M. and Soumare, A.: *The simple and multiple job assignment problems*. International Journal of Production Research 38(14), 3165–3179, 2000.

[8] Chen, Z.; Yang, M.; Francia, G. and Dongarra, J.: *Self Adaptive Application Level Fault Tolerance for Parallel and Distributed Computing*. ipdps, pp.414, IEEE International Parallel and Distributed Processing Symposium, 2007.

[9] Erman, L.; Hayes-Roth, F.; Lesser, V.R. and Reddy, D.R.: *The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty*. Computing Surveys, 12(2):213-253, 1980.

[10] Franklin, S. and Graesser, A.: *Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents*. Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[11] Hayes-Roth, B.: *A blackboard architecture for control*. Artificial Intelligence, pp. 251-321, 1985.

[12] Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P.: *Optimization by Simulated Annealing*. Science, Volume 220, Number 4598, 13 May 1983, pp. 671680.

[13] Koole, G.: *Call Center Mathematics: A scientific method for understanding and improving contact centers*, http://www.cs.vu.nl/~koole/ccmath/book.pdf , 2006.

[14] Millán-Ruiz, D. and Hidalgo, J.I.: *A Memetic Algorithm for Workload Distribution in Dynamic Multi-Skil Call Centres*. Proceedings of the 10th European Conference on Evolutionary Computation in Combinatorial Optimisation (EVOCOP 2010), p. 178-189, Istanbul, Turkey, April 7-9, 2010.

[15] Millán-Ruiz, D. and Hidalgo, J.I.: *Algoritmo memético paralelo para la distribución de esfuerzo en centros de llamadas dinámicos multiagente y multitarea*. (Accepted) To appear in the 7th Spanish Conference on Meta-heuristics, Evolutionary Algorithms and Bioinspired Algorithms (MAEB 2010), Valencia, Spain, September, 2010.

[16] Mladenovic, N. and Hansen, P.: *Variable Neighborhood Search*. Computers & Operations Research 24, pp. 1097–1100, 1997.

[17] Özsu, M. T. and Valduriez, P.: *Principles of Distributed Database Systems*. Second Edition, Prentice Hall, ISBN 0-13-659707-6, 1999.

[18] Pacheco, J.; Millán-Ruiz, D. and Vélez, J.L.: *Neural Networks for Forecasting in a Multi-skill Call Centre*. Proceedings of the 11th International Conference on Engineering Applications of Neural Networks (EANN 2009), p. 291-300, London, UK, August 27-29, 2009.

[19] Russell, S.J. and Norvig, P.: *Artificial Intelligence: A Modern Approach*. 2nd ed. Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2, chapter 2, 2003.

[20] Stützle, T.: *Iterated local search for the quadratic assignment problem*. European Journal of Operational Research, Volume 174, Issue 3, 1 November 2006, Pages 1519-1539.

[21] Wooldridge, M.: *An Introduction to MultiAgent Systems*, John Wiley & Sons Ltd, paperback, 366 pages, ISBN 0-471-49691-X, 2002.

AUTHOR'S INDEX