Optimisation and parallelisation strategies for Monte Carlo simulation of HIV Infection

D. Hecquet [a] H. Ruskin [b] M. Crane [b]

[a] *Department of Computing, INSA de Lyon, Villeurbanne, France*

[b] *Modelling and Scientific Computing group, School of Computing, DCU, Ireland*

Abstract

In recent years, the study of the immune response behaviour through mathematical and computational models has attracted considerable efforts. The dynamics of key cell types, and their interactions, has been a primary focus in terms of building a picture of how the immune system responds to threat. Discrete methods, based on the lattice Monte Carlo method, with their flexibility and relative simplicity have often be used to model the immune system behaviour. However, due to speed and memory constraints, large-scale simulations cannot be done on a single computer. Key issues in the shortening of simulation time are code optimisation and code parallelisation. In this paper are discussed optimisation and parallelisation solutions referring to existing Monte Carlo HIV infection simulation code from previous authors [1], [2].

Key words: Monte Carlo Method, Immune System, HIV, Optimisation, Parallelisation, Domain Decomposition

1 Introduction

Basic Immunology:
The understanding of the immune system is a key issue for immunologists to be able to find new treatments to existing and new diseases.
Most of the time for lethal bacteria or viruses, in-vivo experiments cannot be done hence models are built to simulate part of the real-life behavior of the system. The analyse of the simulation results helps finding new behaviour rules that can be used while elaborating new drugs and treatments.
The immune response is the natural defense of the body against foreign substances and cells that invade, causing minor infections or major diseases. From antigen infection to elimination, the immune response is a complicated process that involves the interactions of a variety of immune cells (Macrophages,

responsible for phagocytosis of pathogens, dead cells and cellular debris, Cytotoxic T Cells and Helper T Cells, type of white blood cell or leukocyte which has on their surface antigen receptors that can bind to fragments of antigens, Plasma B cells secrete antibodies which effect the destruction of antigens by binding to them and making them easier targets for phagocytes, Memory B Cells that are formed specific to the antigen(s) encountered during the primary immune response; able to live for a long time, these cells can respond quickly upon second exposure to the antigen for which they are specific., Antibodies, and Viruses) The immune system has been modelled in several ways, considering different sets of cells, along with a set of interactions in case of an infection. See Ref. [3] and [4] for an in-depth immunology introduction.

Cellular Automata:
Cellular Automata (CA) models were first introduced by John von Neumann in the late 1940's [5] to describe elementary units that can reproduce themselves. Cellular Automata are used to mimic the global behaviour of a system from its local rules and have given good results when being applied to emulate biological, chemical or physical systems, Wolfram [6] and Gutowitz [7].
Since then they have been applied to a multitude of different problems, such as heat and wave equations, Toffoli [8], forest fire spread, Green and al. [9]. These are discrete dynamical systems, typically defined on a n-dimentional grid ($n \geq 1$), that are able to describe continuous dynamical systems through a set of simple rules. Lattice CA are characterized by the following properties:

- A discrete lattice of cells.
- A cell state chosen from a finite set of states.
- A neighbourood (set of cells that can interact with each given cell).
- A set of rules depending on the states of the cell and its neigbours.
- The evolution of the cellular state evolve in discrete time steps.

For probabilistic models, the following property is added:

- A rule can depend on a specified probability.

Monte Carlo Method: Monte Carlo method is a statistical simulation method, using random numbers to give approximate solution of mathematical problems. Developed by Von Neumann [5], Ulam and Metropolis during World War Two to model neutron diffusion in fissile material, Hammersley and Handscomb [10]. Since then MC method has been used to model a large variety of problems, from pi estimation to Schrödinger equation eigenvalues approximation and immune system simulation Dasgupta [11], Mannion and al [12].

CA and MC models of HIV infection: In the case of HIV modelisation,

Monte Carlo method is used with CA to mimic immune system behaviour. The CA is updated using MC methods, that is to say at each step a number of sites equal to the total number of lattice cells are randomly updated. This random site update produces a standard deviation for the result, so several MC runs have to be run to give a satisfactory result. A simple MC CA HIV simulation can then be represented in Fig 1.

(basicHIVSimulationLoop.tif)
Fig 1: Basic Monte Carlo loop for HIV simulation.

For typical MC HIV simulation,

- $1 \leq \text{nrun} \leq 100$.
- $1000 \leq \text{nMonteCarloSteps} \leq 10000$.
- $10 \leq \text{allSites} = \text{lattice size} \leq 1000$ or more.

In the case of 2D simulation, the site can have to be updated from 10^5 to 10^{12} times, in 3D, from 10^6 to 10^{15} times. Knowing that in a typical HIV-positive person, up to 10^{10} viruses are produced daily, with 10^8 new cell infections, and 10^7 virus mutations Coffin [13], Ho [14], these figures must not seem oversized. The "allsite" loop presented here is a basic model, in fact the full update of the lattice during a MC step may require more than one loop that increases further the amount of computation to be done.
So in order to be able to run large scale simulation, memory and speed issues need to be discussed. In this paper will be presented some solutions that can be used for optimisating and paralleling existing MC HIV simulation code.

2 Description of the CA models and MC methods considered

Numerous CA models have been built to simulate immune system response to HIV infection. Three Cell models like the ones of Pandey and Stauffer PS1 [15], Pandey P1 [16], Kougias and Schulte KS1 [17] were first formulated considering the Helper cell (H), the cytotoxic Killer Cell (C), and the viral infected cell (V). Each of these cell types is represented by a boolean equation

representing high (1) or low (0) concentration. More complete models include the five cells Pandey and Stauffer model PS2 [18] or eight cell Pandey P2 [19]. Many more can be found in the litterature such as and immunology, de Boer and al. [20], Deffner [21], dos Santos [22], Seiden and Celada [23].
In this article, the number of cells used by the model will be called nCell and the lattice side size l. The CA and MC models of HIV infection from Mannion [1] and Liu [2] are considered in this article, as their optimisation and parallelisation are required in order to increase the scale of the simulation and were the subject of the internship of one of the authors.
The characteristics of Mannion's model are:

- MC/Stochastic simulation for immune response.
- 2 dimensional lattice.
- Four cell-types are considered:

(1) Macrophages (M)
(2) Helpers (T4) (or TH1)
(3) Cytotoxic T-Cells (T8 or CT)
(4) Virus (V or AG)

- Von Neumann 2D 4-cells neighbourood.
- Periodic boundaries.
- Cell growth simulated by a logical-ORing between the same cell types.
- Intra-site (inter-cell) Interactions, set of 4 equations:

(1) M' = M or V
(2) H' = (M or H) and (not V)
(3) C' = H and M and V
(4) V' = ((V or M or H) and (not C))

- Mutation: probability `pmut` is introduced in equations 1 and 3 to simulate the probability of an antigen having mutated and escaped recognition.
- Mobility: each cell can move from one site to another after interaction, with a fixed probability.

Mannion's code is written in Fortran 77.
The characteristics of Liu's model are:

- MC/Probabilistic simulation for immune response
- 3 dimensional lattice.
- Nine cell-types are considered:

(1) Macrophages (M)
(2) TH1 cells (T1)
(3) TH2 cells (T2)
(4) Cytotoxic T-Cells (CT)
(5) Memory T-Cells (MT)
(6) B-Cells (B)
(7) Memory B-Cells (MB)
(8) Antibodies (AB)

(9) Antigens (AG)
- Von Neumann 3D 6-cells neighbourood.
- Periodic boundaries.
- Growth, logical-ORing between the same cell types. For each type of cell, growth is controlled by a probability fixed for each cell type.
- Intra-site (inter-cell) Interactions, set of 7 equations:

(1) M' = M or AG and (not(M and AG))
(2) T1' = M and not(T1 or AG)
(3) T2' = T2 or ((AG and M) and not(T2 and AG))
(4) CT' = AG and (M or T1)
(5) B' = B or (T2 and AG)
(6) AB' = AB or ((B or M) and AG)
(7) AG' = (M or AG) and not((CT or T1) or AB)

Each one of those is controlled by independent probabilities.
- Death of cells is considered and controlled by a probability fixed by relative cell life expectancy.
- Mutation: probability pmut for an antigen to have mutated and escaped recognition.
- Mobility: each cell can move for a site to another after interaction, controled by a fixed probability

Liu's code is written in C++.
The principles of these algorithms are similar, as is shown in the generic scheme of a HIV MC simulation Fig. 2 (Mannion's model does not consider cell death).

(HIVSimulationLoop.tif)
Fig 2: Monte Carlo and Cellular Automata HIV simulation (3D case shown).

3 Choices of Optimisation Technique

In this part will be discussed optimisations that can be done on existing code without rewriting. Of course these are not universal rules as optimisation has proven to be implementation-dependant, each particular problem being optimisable in a different way. Even if hardware-optimised code in assembly langage were to be the best solution to optimise code in our case, this issue

will not be discussed here, neither cache optimisation as the memory space needed during the execution exceeds the size of the cache.
First to be considered are the non langage dependent optimisations, that can almost be inplemented on every kind of existing code.

Memory Optimisation:
Consists in data storage optimisations:
To store the l size lattice with n cell types, three solutions can be implemented:

- Storage in n boolean l size lattices.
- Storage in a single boolean (l*n) size lattice indexing by cell type
- Storage as a l size char matrix each bit of the char representing a cell type, bitwise operations are used during computation. Char can be used if $n \leq 9$, then integers can be used. Bit-strings can also be used in C, making the code far more readable and preventing from having to index as boolean lattices as considered cell types.

It should be remembered that most of the time memory issue is less important than speed issue because increasing the size of the lattice will cause the time taken by the simulation to increase faster (from a few days to a few weeks) than memory taken to make the simulation. Current computer available memory allows large computations that can take days, so speed optimisation is more important.

Speed Optimisation:
Algorithmic optimisations:

- Lookup table use:
  Considering Nc cells following boolean equations of Np parameters. The update calculus can be replaced by addressing a 2Nc+Np lookup table pregenerated during compilation if Nc+Np not too big (experimentally ≤ 20).
- Conditional loop reduction:
  If then else statements containing boolean updates can be directly integrated into calculus. if (a) then (b = boolean_calculus1) else (b = boolean_calculus2) can be replaced by b=(a and boolean_calculus1) or (not(a) and boolean_calculus2)
- Loop optimisation:
  Using loops increase execution time as the program spend time in incrementing and testing loop index. For small loops, polynomial evaluation using Horner's Method is faster, ie eval=((a(0)*X+a(1))*X+a(2))*X+a(3) is faster than

```
eval = a(0);
for(i=1;i ≤ 3;i++){
        eval = eval*X + a(i);
}
```

- Loop unrolling:
  For bigger loops Loop Unrolling can be used and consists in treating sequentially fixed-size loops in order to get rid of loop initialisations and closings if the resulting code size increase is acceptable.

```
for(i=1;i ≤ 100;i++){          1 init.
    F(i);                      100 exec.
}                              100 closes
```

  is slower than

```
for(i=1;i ≤ 50;i+2){           1 init.
    F(i);                      50 exec.
    F(i+1);                    50 exec.
}                              50 closes
```

  Depending on the size of the loop this optimisation revealed to be efficient.
- Loop rearrangement:
  If loop unrolling cannot be done, loop rearrangement can reduce the number of loop initialisations and closings:

```
for(i=1;i ≤ 20;i++){                    1 init.
    for(j=1;j ≤ 10;j++){                20 init.
        for(k=1;k ≤ 5;k++){             200 init.
            loop code                   1000 exec.
        }                               1000 closes
    }                                   200 closes
}                                       20 closes
```

  This code gives a total of 221 loop initialisation and 1220 loop closings. If possible, this can be reduced by rearranging the loop as

```
for(k=1;k ≤ 5;k++){                     1 init.
    for(j=1;j ≤ 10;j++){                5 init.
        for(i=1;i ≤ 20;i++){            50 init.
            loop code                   1000 exec.
        }                               1000 closes
    }                                   50 closes
}                                       5 closes
```

  For a total of 56 loop initialisation and 1055 loop closings, we save 165 loop initialisations and 165 loop closings. With small changes in the loop code if the index order matters, the result will be the same as the loop code is

executed the same amount of time but time is saved by having less loop initialisations and closings.

As Fortran and C/C++ code were considered, langage dependent optimisations have to be considered, for each of these langages.

- Column and row major order:
  In Fortran, arrays are stored in memory by column, ie in the order a(1,1), a(2,1), , a(n,1), a(1,2), a(2,2), , a(n-1,n), a(n,n). In C, row major order is used. Paying attention to this ordering can affect performance as access to memory zones is faster when done jumping from a block to its immediate neighbours. In both cases, loop interchange can be necessary to have the smallest stride possible, and respect the array ordering of the language.
- Lattice declaration in C/C++:
  There is two basic ways of storing a n+1-dimension lattice of integers (n dimensions plus cell type dimension). Consider the following two pieces of code, in 3D case:

```
int*** intVector;
intVector = new int**[Mi];
for (i=0;i¡Mi;i++)
{
        intVector[i] = new int*[Mi];
        for(j=0;j¡Mj;j++)
        {
                intVector[i][j] = new int[Mk];
        }
}

int* intVector2;
intVector = new int[Mi*Mj*Mk];
```

  Where Mi, Mj and Mk stand for the maximum values of i, j and k.
  In the first case the vector created is then accessed using intVector[i][j][k].This method is far from being the fastest as it requires three memory accesses for each site as a cell represented by an int***. The other way is accessed using indexing intVector2[i + j*Mi + k*Mi*Mj] and saves time as multiplications and additions are far more efficient than memory access.
- C compiler optimisation, from -O1 to -O5, usually -O3 gives good results, and -Os if the executable can then be stored entirely in the cache.
  · O1: local optimisation.
  · O2: global optimisation.
  · O3: more global optimisation.
  · O4: Interprocedure analyse.
  · O5: Loop unrolling and software pipelining.
  · Os: Size optimisation.
  See Man pages of the gcc compiler for further information.

- C Inlining : this consists of replacing each procedure call by the code of the procedure. For short procedures being called many times, this suppress the overhead due to procedure call (register save, call, return procedure), makes loops and scalar optimisation easier but increases the compilation time, as register allocation is more complex and the code size is increased.
- Low level trick for C loops: replacing
  for(int i=0; i ≤ n; i++) {loop} by for(int i=0; i ≤ n; ++i){loop} saves memory space. As the post-fixed version i++ increments i, then creates a copy of the previous i to use in the loop, copy deleted at the end of the loop, at each step of the loop a copy is created and deleted. The prefixed version ++i prevents this useless allocation.

4 Choices of parallelisation techniques

A look back at the Monte Carlo loop already gives clues on how to parallelise the computation:

```
do nrun
        init Lattice
        do nMonteCarloSteps
                do allSites
                        xsite update
                done
        done
done
```

Due to the independence of each run, each one being done on a differently seeded lattice with a different sequence of random updates, the nrun loop can be made parallel. It is relatively easy to do, giving each process a number of runs to execute, then gathering the data of each computation at the end. The nMCSteps loop cannot be parallelised as each step is sequentially dependent of the previous one using its results.
The allSites loop can be parallelised giving each process a part of the lattice to proceed. This require the choice of an efficient domain decomposition. So two different ways of parallelising a Monte-Carlo simulation have been studied:

- Time parallelisation, giving each process a certain number of MC runs to do.
  · Advantages:
    It only generates a little communication overload for synchronisation during initialisation and for data gathering at the end of the computation.

It gives a predictible and good speedup, typically with nProc process and nRuns to process, the computation takes a little more than (single proc time) / min(nProc,nRuns) if nProc $\geq$ nRuns
(single proc time) / nRun * $\lceil$ Runs/nProc $\rceil$ if nProc $<$ nRuns
where $\lceil$ x $\rceil$ is the nearest upper x integer.

· Disavantages:

When even a single run takes a lot of time to process, huge lattices that are treated slowly by a single process gives little increase in speed.

This solution is quite easy to implement using the simplest parallel methods and algorithms. MC : Ref o_O

- Spatial parallelisation, separating the lattice domain in stripes (such as the domain decomposition used by Shu and al [24] and fig. 3), each one of them given to a process that makes the update of its zone.

  · Advantages:

  For big lattices, the memory needed is divided among all processes. This allows the computation of bigger lattices.
  Speedup effective on each run.

  · Disadvantages:

  High communication overload, for highly interacting system can give no speedup. The more interactions a single MC step contains the more communication is needed and the slower is the program.

The updates done at the border of two domains are propagated to the neighbour. All processes can be treated equally sending its changes to its neighbours and receiving asynchronously the changes when it needs to update a border site, using symetric code. A master process can gather all changes done and distribute data when requested to by a process updating a border site. Fig.3 shows how the domain is decomposed with each process having a copy of the border sites of its neigbours (ghosts points), copy updated each time needed. This method creates an overhead in communication and in memory use as some parts of the lattice are stored twice.

(DomainDecomp1.TIF)
Fig. 3: 2D domain decomposition, each processor communicates with its upper and lower neighbours.

5 Results and Evaluation of Parallelisation and Optimisation of considered code

Optimisation: To get an estimate of optimisation efficiency, we use a form of speed up measure, simply comparing execution time of original code and optimised codes:

$$S = \frac{T_{orig}}{T_{opte}} \tag{1}$$

In the case of Mannion's [1] the code optimisation techniques used are the following:

- Code was rewritten in C, in order to be able to use more optimisation techniques.
- Compiler optimisation -O3 was used.
- 4 boolean l^2 lattices were replaced by a single char l^2 lattice, using bitwise operations to select cell types.
- Update equations were replaced by two 2^4 lookup tables, one in case of mutation, the other with mutation, a single 2^5 lookup could have been used but the other solution was implemented for simplicity's and readability's sake.
- Loop reordering was changed to respect C row major order.
- Some loops on the cell type were unrolled.
- Multiplicative Linear Congruential Random Number Generator routine was inlined.
- Prefixed loop index was used.

The use of these methods give good results due to the relatively small number of cell types and interaction equations. Speedup for Mannion's code with the above techniques can be seen in fig. 4.

(R_MannionOptim.TIF)
Fig. 4: Results from the optimisation of Mannion's code.

Previously written using Microsoft®Visual C++®with MFC®Graphics [25] and the SDK®platform [26], Liu's code [2] first had to be written again to be usable on a linux cluster. Once the optimisation done, the results on execution time were quite surprising. For relatively small lattices (of size ≤ 100) optimisation techniques used give no observable speedup compared to original code using compiler optimisation. The main reason is that for these lattices with nine cell types and a lot of equation parameters (a total of twenty three variables to take in account for each site update), the speed up given by addressing a single lattice instead of nine is cancelled by the time taken by index calculation and bitwise operations.
For larger lattices (size ≥ 100) the number of memory accesses in the nine lattice original code gets bigger and the single lattice code finally computes faster, with a 33% increase in speed at the end, cf Fig. 5 and 6.

(Y_LiuOptim1.TIF)
Fig.5: Results from the optimisation of Liu code big scale.

(Y_LiuOptim2.TIF)
Fig.6: Results from the optimisation of Liu code small scale.

Parallelisation: For both pieces of code, code parallelisation was realised using the Message Passing Interface (MPI) standard [30], [28], [29], and the MPICH implementation [27]. For simplicity's sake, the use of basic MPI primitives was privileged as all MPI code can be implemented using MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Barrier, MPI_Reduce, MPI_Finalize, MPI_Send and MPI_Receive.
In order to have an estimation of the parallelisation result for n processes, we use the common performance evaluation that are speedup S_n and efficiency E_n respectly defined as

$$S_n = \frac{T_1}{T_n} \qquad and \qquad E_n = \frac{S_n}{n} \tag{2}$$

Both time and spatial parallelisation were implemented for Mannion's code [1]. In our case, as we have to run the simulation several times as Monte Carlo methods has a standard deviation linked to the random generator used, the most efficient method is the time parallelisation. In this case the speed up is optimum if the number of runs to proceed is a multiple of the number of avalaible processors. With 10 processors and 10 runs, the speedup is between 9 and 10.
The spatial parallelisation gives little speed up, at least compared to the previous method results. This result is essentially due to the fact that the HIV simulation used has high interaction rates, and produces a high communication overhead.

In order to predict this overhead, the easiest way is to count the average number of basic send and receive operations done each MC step. Each process is in charge of a $\lceil l/(n+2) \rceil$ domain (on average), including ghosts points. Communication is needed during nearest neighbour interactions when a site at the border of a domain is updated, as the update has to be transmitted to the neighbour ghost points. Refer to Fig.7 for the names used in the following part. Each part of this domain has a different communication need; communication is needed each time a first level border site has one of its cells that moves or a second level border site cell moves towards a first level border site. So, on considering the selection of site to update truly random, the communication needed can be undervalued per process as follows (considering a master process gathering all changes at the domains borders, which represent an optimised solution in terms of communication overhead):

- Nearest neighbour interactions:
  - Each first level border site update needs a check for a ghost point change (under the form of a send and receive operation with the master process in terms of communication).
  - Once nearest neighbour interactions are computed, global update needs a send operation to the master.
  - No communication is needed for second level border sites.
- Mobility: in this case, as a process can modify the first level border sites of its neighbours while moving a
  - cell from its first level border sites to its ghosts sites, more communication is needed.
  - Each first level border site update needs the check of surrounding zone changes (a send and receive operation with the master).
  - Each first level border site update needs a send operation done if communication method is optimised.
  - quarter of the second level border site update needs to be checked for a first level border site change (a send and receive operation with the master) for each cell type.
  - quarter of the second level border site update needs a send operation for

each cell type.

In all cases the master keeps waiting for send operation from the slave processes, operation whose flag tells it what to do.
Each MC step Nearest Neighbour Interactions are done l^3 times, and mobility $nCell*l^3$ times. Summed for all processes, for a MC step a rough estimate of communication overload is:
For NNI, per slave:
Send: 2*l + 2*l
Receive: 2*l
For Mobility, per slave:
Send: 2*l + 2*l +2*l + 2*l*1/4 + 2*l*1/4
Receive: 2*l + 2*l*1/4
For a total of a little less than 16*l send and receive calls per processor each MC step, in the best case, as we simplified the mobility case.

(DomainDecomp2.TIF)
Fig.7: Single processor domain decomposition.

In the case of Liu's code [2] only time parallelisation was implemented, as spatial parallelisation in 3D case the 3D modelisation that increased communication overhead. Time parallelisation gave the same results as the 2D case, execution time roughly divided by the number of process.

An attempt of software emulated shared memory program was implemented. This implementation was used to allow each process to write the changes done in their domain directly their neighbour's shared memory. This gives low speedup results, telling us that non hardware supported shared memory is to be avoided.

6 Conclusion

The objective of the optimisation and parallelisation of HIV infection simulation code is to find the optimal ways to reduce computation time of CA and MC simulations. While this article is not meant to give general optimisation and parallelisation techniques for biocomputational simulations it does reveal for parallel execution of CA and MC code the most effective parallelisation method. Methods using spatial parallelisation have a big communication overload as the exchanges done to update each process domain during computation are time-consuming. The results of these techniques can also be applied to MC and CA simulation that need to compute several runs then aggregated to deal with the standard deviation of MC methods.

7 Acknowledgements

The work described in this paper was supported by a grant from the National Instutite for Cellular Biotechnology (NICB). I would like to thank my supervisors Heather Ruskin and Martin Crane for all their patience and advices.

References

[1] Mannion, R. 2001.CA and Monte Carlo Models of HIV Infection. M.Sc. Thesis, Dublin City University, 2001.

[2] Liu, Y., 2002. A Mesoscopic Approach to Modelling the Immune Response Using Cellular Automata and Monte Carlo Methods. M.Sc. Thesis, Dublin City University, 2002.

[3] Kuby, J., 1992. Immunology, New York Freeman.

[4] Roitt, I.M., 1994. Essential Immunology 8th ed. Oxford; Boston : Blackwell Scientific Publication.

[5] Neumann, J.V., 1966. Therory of self-reproducting automata. Ed. A.W. Burks, University of Illinois Press.

[6] Wolfram, S, 1994. Cellular Automata and Complexity: Collected Papers. Addison-Wesley

[7] Gutowitz , H., 1991. Cellular Automata: Theory and Experiment. Physica D45 (1990) Nos. 1-3, and MIT press book.

[8] Toffoli, T., 1994. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. Physica D: Nonlinear Phenomena, Volume 10, Issues 1-2, January 1984, pages 117-127.

[9] Green, D.G., Tridgell, A., and Gill, A.M., 1990. Interactive simulation of bushfire spread in heterogeneous fuel. Mathematical and Computer Modelling 13 (12), 57-66.

[10] Hammersler, J.M. and Handscomb, D.C., 1964. Monte Carlo Methods. Chapmann and Hall, New York.

[11] Dasgupta, S., 1992. Monte Carlo simulation of the shape space model of immunology. Physica A: Statistical and Theoretical Physics, Volume 189, Issues 3-4, 1 November 1992, pages 403-410.

[12] Mannion, R., Ruskin, H.J. and Pandey, R.B., 2002. Effects of Viral Mutation on Cellular Dynamics in a Monte Carlo Simulation of HIV Immune Response Model in Three Dimensions. Theory in Biosciences, Volume 121, Issue 2, pages 237-245.

[13] Coffin, J., 1995. HIV population dynamics in vivo: Implications for genetic variation, pathogenesis, and therapy. Science 267, 1995, 483-488.

[14] Ho, D.D. 1996. Viral Counts Count in HIV Infection. Science 272, pages 1124-1125.

[15] Pandey, R.B. and Stauffer, D., 1989. Immune Response via interacting three dimensional network of cellular automata. Journal of Physics, 50, pages 1-10.

[16] Pandey, R.B., 1989. Computer Simulation of a cellular automata model for the immune response in a retrovirus system. Journal of Statistical Physics, 54, pages 997-1009.

[17] Kougias, C.F. and Schulte, J., 1990. Simulatin the Immune Response to the HIV-1 Virus with Cellular Automata. Journal of Statistical Physics, 60, pages 263-273.

[18] Pandey, R.B. and Stauffer, D., 1990. Metastability with probabilistic cellular automata in an HIV infection. Journal of Statistical Physics, 61, pages 235-340.

[19] Pandey, R.B., 1991. Cellular automata approach to interacting network models for the dynamics of cell population in an early HIV infection. Physica A, 179, pages 442-470.

[20] De Boer, R.J., Segel, L.A. and Perelson, A.S., 1992. Pattern Formation in One and Two-dimensional Shape-space Models of the Immune System. Journal of Theoretical Biology. 155, pages 295-333.

[21] Deffner, M., 1993. Computer Simulation Of Immunological Cellular Automata: the Tsallis Model. Physica A, Volume 195, pages 279-285.

[22] Santos, Z.D. 1993. Transient times and periods in the high-dimensional shape-space model for immune systems. Physica A: Statistical and Theoretical Physics, Volume 196, pages 12-20.

[23] Seiden P.E. and Celada, F., 1992. A computer model of cellular interactions in the immune system. Immunology Today, Volume 13, Issue 2, pages 56-62.

[24] Shu J.L., Lu Q., Wong W.O., Huang H., 2001. Parallelization strategies for Monte Carlo simulations of thin film deposition. Computer Physics Communications 144 (2002) pages 34-45.

[25] Microsoft®Foundation Class Library. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/mfchm.asp

[26] The Microsoft®Windows®Software Development Kit (SDK) http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en

[27] MPI implementation, http://www-unix.mcs.anl.gov/mpi/mpich/

[28] Gropp, W., Lusk E., Skjellum, A., 1999. Using MPI - Portable Parallel Programming with the Message-Passing Interface second edition. The MIT Press.

[29] Gropp, W., Lusk, E., Thakur, R., 2000. Using MPI-2 - Advanced Features of the Message-Passing Interface. The MIT Press.

[30] Message Passing Interface, http://www-unix.mcs.anl.gov/mpi/