Parallel Monte Carlo Simulation of MBE Growth

Isabel M. Beichl
National Institute of Standards & Technology
Gaithersburg, MD 20899

Y. Ansel Teng
Jet Propulsion Laboratory
Pasadena, CA 91109

James L. Blue
National Institute of Standards & Technology
Gaithersburg, MD 20899

Abstract

We present a parallel algorithm for Monte Carlo simulation of molecular beam epitaxial growth (MBE) focusing on the software we have developed and the experiences gained. An efficient sequential method to do MBE simulation was developed that uses a single binary tree to store the rates of occurence of all possible events at a given time. The challenge was to find a way to adapt this method to a parallel machine.

1 Introduction

In this paper we present a parallel algorithm for Monte Carlo simulation of molecular beam epitaxial growth (MBE) using the Connection Machine, CM-5. This paper focuses on the software we have developed and the experiences we have gained. An efficient sequential method to do MBE simulation was developed that uses a single binary tree to store the rates of occurence of all the possible events at a given time. The challenge was to find a way to adapt this method to the parallel machine.

We are interested in using parallel processing for this simulation for two reasons: on one hand, Monte Carlo simulation usually requires a large number of repetitions to derive meaningful statistical data, thus it can be a very lengthy process; on the other hand, the problem instances we are interested in can be too large to be handled efficiently by a single-processor computer: since the simulation is performed at the atomic level, a sample may contain trillions of elements to be simulated.

These two reasons lead us to two different approaches in coping with this problem. The first approach is a straightforward parallelization of the repetitions. Each processor will perform the simulation on its own problem instances. The processors communicate only at the end of the entire simulation to collect the statistical data. The second approach aims to have all the processors cooperate to complete a single problem instance so that a very large problem instance can be performed efficiently. We take a domain-decomposition approach by dividing the grid into subgrids and assign one processor to each subgrid.

In the following sections we will discuss our algorithms and implementations for each of these approaches. The programs are written in C with a hostless execution mode using the CMMD communication library for interprocessor communications. That is, each processor node will execute the same C program and the control node will only run a system-provided routine to start the nodes and then serve as an I/O server. These language constructs provide full asynchrony at each node and the possibility for exploiting the node architecture using lower-level languages in the future.

An important factor in a parallel Monte Carlo simulation is that each processor should use a sequence of pseudo-random numbers that are random not only in each sequence but also across the processors. Currently, we use a multi-processor lagged-Fibonacci random number generation system developed at the Supercomputing Research Center [1].

2 Explanation of the problem.

In our MBE growth simulation, the sample is modeled as a two-dimensional regular grid. Each grid cell has a number of atoms accumulated, defined as its *height*. New atoms are sprayed on top of the sample, while atoms on the top layer may move to one of its neighboring locations. The goal of our project is to model the growth of the sample in such a situation. An event, then, can be a deposition or a diffusion (the top atom will hop either north, south, east or west), the rate of deposition being constant and the rate of diffusion an event at each location being determined by the geometry at that location and the global physical condition. Therefore, when an event occurs, in the case of deposition, the rates for that site must be updated as well as for the 4 neighboring sites and, in the case of diffusion, the 2 affected sites as well as the 6 neighboring sites must be updated.

In a sequential setting, the rates of deposition and diffusion for each site in the grid are stored in a binary tree whose leaves contain the information and whose nodes contain the sums of the 2 descendents. This is described in [3]. The simulation routine repeats a loop that first determines which event should occur, and then carries out the event by modifying the heights of the grid cells accordingly. The event is determined by selecting a random number between 0 and the sum of all the tree elements which is available at the root of the tree, and locating the corresponding leaf in the tree. The information at the leaf corresponds to an event.

3 Approach I: Independent Instances

The simulation operates on two fundamental data structures: one is the height of the grid cells, *height[n][m]*, configured as a two-dimensional array of n rows by m columns. The other is the *event tree*, which organizes the rates of all the events in a tree structure. At each iteration, a random number between 0 and the sum of all event ratesis generated. Using the event tree, we can efficiently choose an event to be carried out on the grid.

In the version I software, parallelism is exploited by independent execution of different problem instances. Except for some top-level controls, the simulation routine is exactly the same as the sequential program.

4 Approach II: Domain Decomposition

As we mentioned in the introduction, the second reason for using parallel processing for the MBE growth simulation is to handle very large problem instances that traditional computers may not be able to handle efficiently. Since memory size is the primary concern in this situation, we must divide the data structure and store in each processor a portion of the entire problem. Therefore, we take a domain decomposition approach that divides the grid into uniform subgrids and assign one processor to each subgrid.

Ideally, if we use p processors for a problem instance, we should obtain a p-fold speedup if all the processors work independently on their own subgrid. However, the following considerations complicate the problem:

1. Global consistency: an event initiated in a subgrid may involve a cell in a neighboring subgrid, or will change the rates of events in neighboring subgrids. The neighbors must be notified and update their subgrids to maintain global consistency.
2. Conflicting events: since an event may cause diffusion across the boundary of its subgrid, it can conflict with other events initiated by the neighboring subgrids. We must resolve these conflicts according to physical laws and probability theory.
3. Synchronization: the frequency at which events occur in each subgrid should be determined by the total rate of its events, rather than the processing speed of the corresponding processor.

Based on these considerations, we designed the following simulation scheme. The grid is partitioned into subgrids. Each processor is assigned a subgrid, called its *territory*. Since the event rates of a cell are determined by its neighborhood, each processor will also keep the heights of those neighboring cells outside its territory. We refer to these cells as its *outer boundary*, as opposed to its *inner boundary*, which contains bordering cells in its territory. More specifically, if the territory contains grid locations between rows n_0 to n_1 and columns m_0 and m_1 then the outer boundary are the locations in row $n_0 - 1$ from columns $m_0 - 1$ through $m_1 + 1$, and row $n_1 + 1$ from columns $m_0 - 1$ through $m_1 + 1$, and the locations in column $m_0 - 1$ from rows $n_0 - 1$ through $n_1 + 1$ and in column $m_1 + 1$ from rows $n_0 - 1$ through $n_1 + 1$. We also define the *source* of an event to be the cell chosen by the event tree, and the *destination* to be the cell whose height is changed by the event that occured at the source. In a deposition, the source and the destination are identical. The *participants*, or *involved cells*, are the source or the destination. The *affected cells* refer to the neighbors of the participants whose rates may need to be updated as a consequence of the event. In the execution of an event, only the participants will have changes in height, while both the participants and the affected cells may have changes in their event rates. These definitions are also extended to subgrids. Each processor will maintain an event tree only for cells in its territory,so the event initiated by a processor must have its source cell in its territory.

The simulation routine is a loop consisting of the following components:

1. `random_event()`: determine a random event using the event tree. In order to maintain appropriate relative growth rates across all subgrids, a normalization process is applied by generating at each processor a random number between 0 and the maximal total event rate over all subgrids. If the value is smaller than the total rate of its subgrid, an event will be located in the event tree. Otherwise, the processor will initiate no events at that iteration. This normalization operation also synchronizes all processors at the beginning of each iteration.

2. `check_conflict()`: collect the neighboring events that may affect the territory, check if there are any conflicts and determine which events should occur.

3. `commit_event()`: carry out the list of events that survive the conflict checking. For each event, the processor updates the heights for cells within its outer boundary, but only updates the rates and the event tree for cells in its territory.

4.1 Conflict Resolution

A major challenge in Version II software is the design of an efficient communication protocol for the conflict resolution process. In this section we give more detail about this process.

In the `check_conflict()` routine, each processor should receive neighboring events that may involve cells in its subgrid, including its outer boundary, and determine which events should be carried out. This is a typical mutual exclusion problem that can be represented by a conflict graph and modeled as a graph coloring problem.

Since our goal in this approach is to run the simulation on very large grids, we expect few conflicts among the subgrids. So we adopted a local maximal criterion for a fast and globally consistent resolution. When a conflict occurs, it is resolved by determining which event occurred "first". For each subgrid, a measure of time is kept as in [2]. This number is part of the event description sent to the affected neighbors. The processor with the earlier event will have priority over other conflicting events. If an event has an earlier *time* than all other events that are in conflict with it, this event will remain valid and the others will be discarded.

Since communication occurs only between logical neighbors, the virtual channel functions in the CMMD library were the most suitable for our implementation. The virtual channel provides a logical channel between a memory location in the source processor and a memory location in the destination processor.Once the channel is established, messages can be sent over the channel one at a time with low latency. Normally, a channel is in either a writable or a readable status. After a channel is opened, it has a writable status, and the source can send a message over the channel. When the send operation succeeds, the channel will be in a readable status, indicating that the data is available at the destination location. Then the processor at the receiving end can reset the channel to a writable status so that the sender can transmit again.

The straight-forward solution for conflict checking is to have each processor send its event to all its eight neighbors and receive all neighboring events in a synchronized manner. It will then determine among all events affecting its subgrid which ones will be committed. This solution is expensive in term of both its communication time and its computation time.

As we expect few communications and even fewer conflicts among large subgrids, we would like to use an asynchronous communication scheme that sends and receives a minimal number of messages only to relevant neighbors. So each processor sends its event to neighbors that might be affected. Then it collects relevant events from neighbors and checks if there are conflicts among its own events and those from the neighbors. The checks are performed upon receiving each neighboring event if its own event is still valid. If its own event is overruled by an incoming event, the processor will send canceling signals to its neighbors to whom it has sent its event in the beginning. Upon receiving a cancel signal from its neighbor, a processor will mark the corresponding event canceled.

To implement this algorithm, a processor opens four channels for each of its eight neighbors: one for sending its own event, one for receiving the neighboring event, one for sending its cancel signal, and one for receiving the cancel signal from the neighbor. To make sure that all processors keep checking their incoming messages until all communications are completed, we use the global OR flag provided in the CMMD library to coordination all processors. Figure 1 is an abstract of the protocol for each processor.

A key point in this protocol is to hold the channel whose event triggers the canceling signals until the canceling signals are received. When a processor sets the global OR flag to 0, it has finished not only the communication of its own event, but also all the canceling signals due to its event. The processor will remain in the while loop until all processors have set the OR flag to 0. This ensures that all communications have been completed before any processor can leave

```
Algorithm check_conflict()
begin
set global OR flag to 1;
synchronize with all processors;
for each neighbor
        if my event may affect the neighbor
            send my event to the neighbor;
while the global OR flag is 1
        if there is a readable incoming event channel
            if my event is still valid and overruled by incoming event
                send cancel signals to all affected neighbors;
                mark my event as canceled;
                mark the event channel as on-hold;
            end if
            if the incoming event channel is not on hold
                reset the channel
        end if
        if there is an incoming cancel signal
            mark the corresponding event as canceled
            reset the channel
        end if
        if all cancel signals sent have been received
            reset the on-hold channel
        if all outgoing event channels have been reset
            set global OR flag to 0;
end while
end Algorithm
```

Figure 1: Protocol for conflict checking.

size	iteration in 1000	deposition ratio (%)	time/iteration (uSec)
32×16	1	3.19	213.0
32×16	10	13.94	206.0
32×16	100	15.79	205.0
32×16	1000	16.55	204.5
64×32	1	6.88	234.4
64×32	10	18.53	226.0
64×32	100	24.45	221.3
64×32	1000	27.40	219.2
128×64	1	20.60	276.6
128×64	10	17.43	282.2
128×64	100	34.02	265.5
128×64	1000	39.83	259.8
256×128	1	57.68	281.0
256×128	10	30.50	313.1
256×128	100	43.53	300.3
256×128	1000	52.67	290.8
512×256	1	92.49	273.6
512×256	10	67.48	304.3
512×256	100	54.63	321.2
512×256	1000	61.43	312.1
1024×512	1	99.48	300.9
1024×512	10	95.27	305.6
1024×512	100	79.27	327.7
1024×512	1000	76.20	333.7

Table 1: Experimental results for the Version I software.

the loop.

5 Experimental Results

Both Version I and the Version II have been implemented on the Connection Machine CM-5 at the UMIACS at the University of Maryland which has 32 nodes. A series of experiments have been conducted to measure performance with various grid sizes and iterations. Table 1 lists the results of the Version I software which uses the independent execution approach. The results shown are the average of the 32 instances. Because each grid cell requires 44 bytes of data, 1024×512 is the largest size we can execute on each node with its 32 Mbytes memory.

In each iteration, an event will be executed, which can be either a deposition or a diffusion. The deposition ratio indicates the percentage of events that are depositions. The average running time of each event is affected by two factors. The first one is the grid size. Since most of the time is spent in updating the event tree, it is not surprising to see that the running time increases in a logarithmic rate in the grid size. The second factor is the deposition ratio. A deposition involves five cells while a diffusion involves nine, so a higher deposition ratio will result in a lower event cost. Throughout our experiments, the default deposition rate of one layer per minute is used, so that the ratio between the deposition rate and the diffusion rates is the same for each cell regardless of the grid size. It is interesting to see that the deposition ratio increases with the grid size. The reason is as follows. An atom standing alone on a flat surface will usually have a very high diffusion rate. In a small grid, the rate becomes the dominant component in the event tree, so it is likely to be chosen for the next few iterations. However, when the grid size is large, the sum of the deposition rate is so large that a stand-alone atom will not have a dominant diffusion rate. Thus more depositions are chosen from the event tree as the grid size grows larger.

Table 2 lists the results of the Version II software which uses the domain decomposition approach. As is the case in Table 1, the number of iterations has almost no effect in per-event measurements once it is large enough. The dominating factor is the grid size. We learned from Table 1 that when the grid size increases, the effect of the event tree height outweights the deposition ratio and results in a logarithmic growth of the computation time. However, in Table 2, we see a decrease in the overall event cost as the grid size increases. Apparently the communication ratio decreases when the grid size is large as the probability of having an event near the boundary decreases. But what is more important is the improved utilization. It is not due to the reduced cancellation, but rather due to the dominance of the total deposition rate in each subgrid. Since each subgrid has the same total deposition rate, all subgrids will have similar total rates when the sizes are large and thus are likely to have a valid event after the normalization. As a result, the Version II software performs quite well for large grids.If we compare the event costs of the two versions at grid size 1024×512 and one million iterations, the domain decomposition approach has already achieved 13.85 times speedup using 32 processors, which corresponds to a 43.3% efficiency.

size	subgrid length	iter 1000	dep	hop	time (sec)	comm. %	cancel	utiliz %	cost uS
32×16	4	1	1611	3959	0.531	61.7	125	17.4	95.3
32×16	4	10	19874	36728	5.22	61.9	1270	17.5	92.8
32×16	4	100	193778	361607	52.5	62.1	12030	17.4	94.5
32×16	4	1000	1925560	3614268	519.1	61.9	117758	17.3	93.7
64×32	8	1	103	4968	0.487	51.4	2	15.8	96.1
64×32	8	10	6754	40167	4.56	51.5	81	14.6	97.3
64×32	8	100	70151	354203	45.0	51.5	690	13.2	106.1
64×32	8	1000	726820	3519462	446.8	51.1	7614	13.3	105.2
128×64	16	1	198	11302	0.540	46.8	2	35.9	46.9
128×64	16	10	6212	55207	4.68	46.9	6	19.2	76.2
128×64	16	100	56552	431881	45.2	46.5	60	15.3	92.5
128×64	16	1000	571589	3897706	446.7	46.2	542	13.9	99.9
256×128	32	1	895	18289	0.706	35.7	0	60.0	36.8
256×128	32	10	20891	119990	6.48	35.9	6	44.0	46.0
256×128	32	100	232526	1006114	63.0	36.1	57	38.7	50.9
256×128	32	1000	2492366	9311510	621.7	35.8	515	36.9	52.7
512×256	64	1	3173	22301	0.659	40.0	0	79.6	25.9
512×256	64	10	39618	208952	5.51	39.7	1	77.7	26.2
512×256	64	100	643255	1630226	62.0	40.3	21	71.0	27.3
512×256	64	1000	7321393	14925141	619.8	40.5	256	69.5	27.9
1024×512	128	1	10967	19049	0.699	36.9	0	93.8	23.3
1024×512	128	10	61018	235781	7.21	36.6	3	92.7	24.3
1024×512	128	100	1113304	1791936	69.7	37.6	5	90.8	24.0
1024×512	128	1000	13311716	15573292	695.8	37.9	47	90.3	24.1
2048×1024	256	1	25063	6623	0.676	33.5	0	99.0	21.3
2048×1024	256	10	147961	167630	7.14	32.7	0	98.5	22.6
2048×1024	256	100	1520086	1605284	70.4	32.6	1	97.7	22.5
2048×1024	256	1000	18225468	12972216	704.9	32.8	9	97.5	22.6
4096×2048	512	1	31336	657	0.621	30.8	0	99.9	19.4
4096×2048	512	10	273580	45712	6.74	29.4	0	99.8	21.1
4096×2048	512	100	2058109	1124474	70.0	28.2	0	99.5	22.0
4096×2048	512	1000	22211023	9609621	698.3	28.7	0	99.4	21.9

Table 2: Experimental results for the Version II software.

6 Future work

Using the vector units in each CM-5 processor node: the current implementation uses only the SPARC chip on each processor node. However, the real computing power of the CM-5 lies in the four vector units on each processor node. We have investigated the possibility of exploiting the vector units for the Version I software. The bottleneck for the Version I software is updating the event tree. It seems very difficult to use the vector units for such operations efficiently given that each vector unit does not have any local autonomy. However, the non-tree methods used in [2] would work better on vector processors.

Disclaimer

Certain commercial equipment may be identified in order to adequately specify or describe the subject matter of this work. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the equipment identified is necessarily the best available for the purpose.

This work was done while author Y.A. Teng was at the University of Maryland, College Park, Maryland.

Acknowledgements

We would like to thank William Carlson, Michael Mascagni, and Francis Sullivan for their help and discussions.

References

[1] M. Mascagni, S. Cuccaro, D. Pryor & M. Robinson, "A fast, high quality, and reproducible parallel lagged-fibonacci pseudorandom number generator" *Supercomputing Research Center Technical Report*, SRC-TR-94-115, 1994.

[2] J. Blue, I. Beichl & F. Sullivan, "Faster Monte Carlo Simulations" to appear.

[3] A.B.Bortz, M.H.Kalos, & J.L.Lebowitz, "A new algorithm for Monte Carlo simulation of Ising spin systems" *J. Comp. Phys.* Vol. 17, pp. 10-18, 1975.