A Fast Implementation of Parallel Discrete-Event Simulation on GPGPU

Janche Sang[1], Che-Rung Lee[2], Vernon Rego[3], and Chung-Ta King[2]
[1]Dept. of Computer and Info. Science, Cleveland State University, OH 44115, USA
[2]Dept. of Computer Science, National Tsing Hua University, HsinChu, Taiwan, ROC
[3]Dept. of Computer Science, Purdue University, West Lafayette, IN 47907, USA

Abstract— *Modern General Purpose Graphics Processing Units(GPGPUs) offer much more computational power than recent CPUs by providing a vast number of simple, data parallel, multithreaded cores. In this study, we focus on the use of a GPGPU to perform parallel discrete-event simulation. Our approach is to use a modified service time distribution function to allow more independent events to be processed in parallel. The implementation issues and alternative strategies will be discussed in detail. We use Thrust, an open-source parallel algorithms library which resembles the C++ Standard Template Library (STL), to build our tool. The experimental results show that our implementation can be more than 60 times faster than the sequential simulation. Furthermore, the speedup curve scales well which indicates that our implementation is suitable for large-scale discrete-event simulation models.*

Keywords: Parallel Simulation, Discrete-Event Simulation, GPGPU, CUDA, Thrust Library

1. Introduction

Discrete Event Simulation (DES) is a widely-used technique that allows an analyst to study the dynamic behavior of a complex system. DES exploits a computer to model a system stochastically at discrete points in simulated time. A simulation program operates on a model's state variables during each of a sequence of time-ordered events and schedules future events during such processing. However, simulation is usually computationally intensive and time-consuming. Typical simulation applications often execute for hours or even days. Therefore, exploiting the availability and the power of multiprocessors to speed up the simulation execution is of considerable interest.

Parallel discrete event simulation (PDES) attempts to speed up a simulation's execution by partitioning the simulation model into components, each of which has its own event set and is executed by a *Logical Process*(LP) on a different processor. To guarantee the distributed events will be executed in an appropriate order, two main types of synchronization mechanisms among LPs have been proposed: conservative and optimistic [1]. Conservative mechanisms do not allow an LP to process an event until it is certain that causality violation will not occur. This means that an LP will not receive an event with a smaller timestamp than its current clock from another LP. However, An LP may wait for events that never arrive. Therefore, LPs may send null messages to other LPs to avoid deadlocks [2]. Optimistic mechanisms ignore inter-process synchronization issues, but make compensations by performing rollbacks to a check-pointed consistent state when a causality error occurs [3]. This requires periodic state-saving of the simulator.

With the advance of graphics hardware technology, programming and executing general applications on GPGPUs is more feasible. Nowadays, a single GPGPU with hundreds or even thousands of processing cores has great potential for improving the performance of various computational intensive applications. In this paper, we focus on the use of a GPGPU to perform parallel discrete-event simulation. Note that the architecture of GPGPU can be classified as Single Instruction, Multiple Data(SIMD). To allow more events to be processed in parallel based on SIMD, our approach is to use a modified service time distribution function which guarantees that the events clustered to be executed simultaneously are independent of each other and hence causality errors will not occur. In other words, our method can be treated as a conservative approach from certain viewpoint.

Our implementation is done with the Thrust [4] on the NVIDIA Compute Unified Device Architecture(CUDA) platform. Thrust is a CUDA library of parallel algorithms with a user-friendly interface resembling the C++ Standard Template Library (STL). It hides the details of low-level CUDA function calls and provides highly-optimized implementation of standard algorithms, such as searching, sorting, reduction, compaction, etc., which greatly enhances developer productivity. Therefore, GPGPU-based applications implemented with Thrust are readable, concise, and efficient.

The organization of this paper is as follows. Section 2 describes related work. In Section 3, an old algorithm which we borrow some ideas from is investigated. Section 4 presents our implementation strategies. In Section 5, the experiments and the results for performance evaluation are presented. We give a short conclusion in Section 6.

2. Related Work

In the area of practical parallel simulation, two apparently orthogonal streams of effort have developed over the past decades. The *replication*-based effort entails natural parallelism and is able to utilize massive data-parallel computational power. The *EcliPSe* toolkit described in [5], [6] has proven to be a very successful system for replication-based simulations. The *distribution*-based effort emphasizes functional decomposition of a model across processors. Examples of systems supporting distributed simulation include ModSim[7], Sim++[8], $ParaSi$[9], and $ParaSol$[10]. An inherent difference between the two approaches is that replication exploits statistical sampling to speed up the generation of multiple (typically, but not necessarily independent) sample paths, while distribution exploits model partitioning to speed up the generation of a single sample path.

Because of its massively data parallel computing power, GPGPU has been used by more and more researchers for simulating large-scale models over the past few years. For example, a discrete-event simulation of heat diffusion performed on GPGPU can be found in [11]. The algorithm selects the minimum among all update times and uses it as a timestep to perform a synchronous update of state across all elements in the grid. Another work reported in [12] focuses on a high-fidelity network modeling and uses the GPU as a co-processor to distribute computation-intensive workloads. Our approach is similar to the work in [13] and [14] which develop an event clustering and execution scheme based on the concept of approximation time. In these two papers, the former illustrates practical implementation strategies, while the latter presents an analysis of the approximation error in their algorithm. Our algorithm borrows some ideas from their algorithm for updating service facilities. The old algorithm will be studied in detail in the next section.

3. The Old Algorithm

The work in [13] and [14] introduced a time-synchronous/event algorithm using a time interval instead of a precise time. Figure 1 shows the pseudo code of the hybrid algorithm. To achieve more parallel processing, their algorithm clusters events within a time interval. That is, the simulation time is divided into many fixed-sized time slots which is similar to the time-based simulation, a methodology usually used for continuous physics/dynamics simulation [15]. However, unlike the pure time-based simulation which advances the time slot by slot, the old algorithm directly moves the clock to the slot which contains the event with the minimum timestamp in the future event set. This could reduce the execution time if a slot doesn't have any events to be processed. Therefore, as shown in Figure 1, all of the events whose timestamps are less than or equal to the time slot boundary (i.e. the smallest multiple of time interval greater than or equal to the minimum timestamp) can be extracted from the future event set and then be executed.

```
while ( current_time  <  simulation_time )
   min_timestamp = find_min(future_event_set);
   current_step = the smallest multiple of time
                  interval greater than or
                  equal to min_timestamp;
   parallel for each event e in future_event_set
       if (the timestamp of e <= current_step)
          extract e from future_event_set;
          process e and generate new events
              into future_event_set;
       end if
   end for
   current_time = current_step;
end while
```

Fig. 1: The pseudo code of the old algorithm

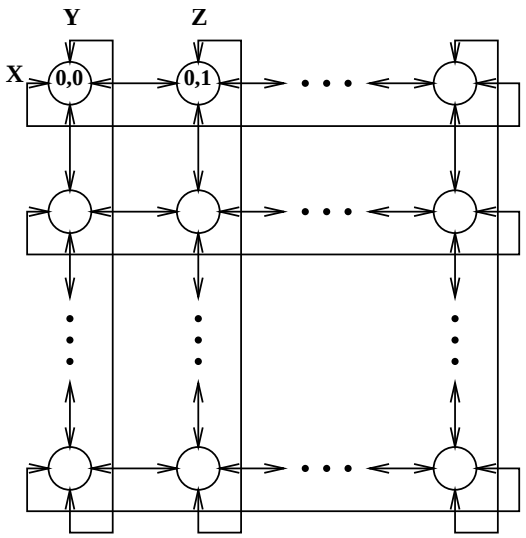
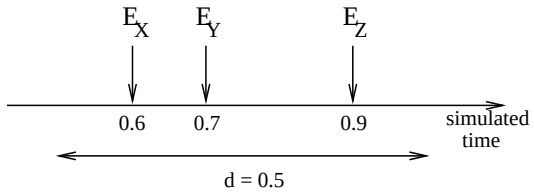


Fig. 2: Torus Queuing Network

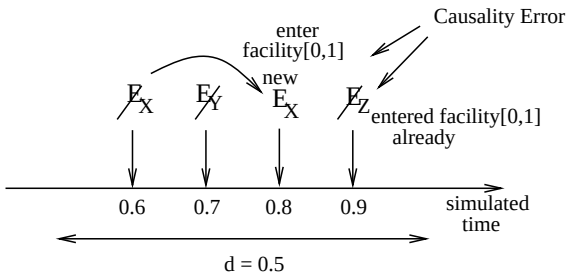
However, the old algorithm cannot be directly used in the precise-time PDES. Note that the PDES should handle the events in a causal consistent way exactly as the sequential DES does. Let's use the simulation of a torus queueing network as an example. As shown in Figure 2, a torus consists of service facilities arranged in a two dimensional mesh. Each facility has four outgoing and four incoming channels. When a token arrives at a service facility, it gets the service for some random amount of time if the server is idle. Otherwise, the token has to wait in the server's waiting queue. After being served, the token moves to one of the four neighbors. For simplicity, we assume that the probabilities of a token leaving a facility on any given outgoing channel are equal (i.e. 0.25).

Assume that there are three tokens X, Y, and Z in the torus network (see Figure 2). The token X and the token Y enter the service facility[0,0] at time 0.6 and 0.7, respectively. The token Z will arrive at the facility[0,1] at time 0.9. Also assume that the service time for the token X being served at



All events within the range d can be process in parallel

(a) before parallel processing



(b) after parallel processing

Fig. 3: Causality Error

the facility[0,0] is 0.2. Using the old algorithm with the time interval $d = 0.5$, all of these three events can be processed in parallel at the time 1.0 (i.e. the smallest multiple of d which is greater than 0.6). The scenario is depicted in Figure 3(a). Note that an event E in Figure 3 represents a combined departure/arrival event.

Since both X and Y enter the facility[0,0], the old algorithm uses the original timestamps to keep the causal order. That is, the token X will get the service immediately, while the token Y will stay in the waiting queue. However, if we use the original timestamp for the token X to calculate its departure/arrival time, the token X should enter the facility[0,1] at time 0.8. As shown in Figure 3(b), a causality error occurs because the token Z, with the arrival time at 0.9, has been served in the facility[0,1] already. Therefore, the old algorithm cannot process the events exactly as the causal order in the sequential DES. We also conducted an experiment to verify this. We recorded the last arrival time for each service facility. If the timestamp of a new arrival is smaller than the last arrival time, a causality error is detected. Figure 4 shows that the larger the interval, the more causality errors occurred in the simulation.

4. The Improved Implementation

Our algorithm for PDES is based on the precise time, not the approximation time as in [13], [14]. The first issue we need to deal with is the potential causality error as discussed in the previous section. To solve the problem, we let the service time for each token contain the constant time interval d and subtract the constant d from the mean service time in

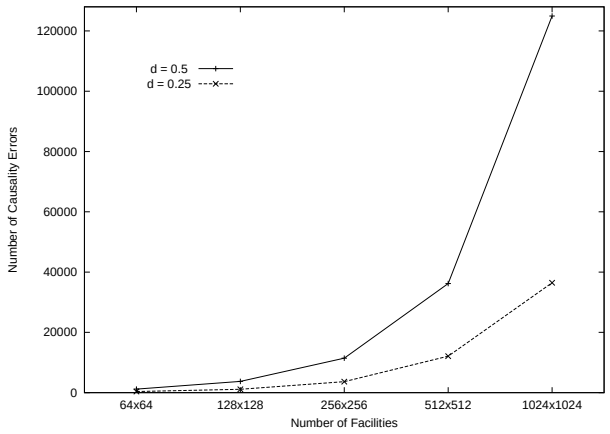


Fig. 4: Causality Errors with varying the number of facilities

the invocation of the service time distribution function. More precisely, if the service time is exponentially distributed, we change the expression of calling exponential distribution function from

```
expon(M)
```

to

```
expon(M-d) + d
```

where M is the mean service time. Note that in the modified formula, the mean service time is still M, but the service time for any token is always greater than d. Therefore, the aforementioned causality error will not occur. For example, the timestamp of the new departure/arrival event for the token X in Figure 3 will be at least $0.6 + d = 1.1$ which is after the token Z enters the facility[0,1].

Another advantage of using the modified formula for the service time is that the full time interval can be used to cluster events for parallel processing. Our algorithm extracts any event which has the timestamp less than or equal to

```
minimum_timestamp + d
```

and hence will include more events than the old algorithm. The more parallel events be executed, the faster program runs. For example, assume that $d = 0.5$ and the minimum timestamp in the future event set is 1.42, the events with the timestamp between 1.42 and 1.50 can be processed concurrently in the old algorithm. The effective range size is only 0.08. Using our algorithm, the range is between 1.42 and 1.92. In general, giving the same interval d, the average effective range size of the old algorithm is half of the range size in our algorithm. However, our method still has its disadvantage. The biased distribution function will yield a small difference as compared with the result of using the original distribution function. The empirical evaluation of the difference will be reported in the next section.

Figure 5 shows our implementation on the host using the Thrust library. As mentioned before, Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). One of the

```
thrust::device_ptr<FACTYPE> all_fac = thrust::device_malloc<FACTYPE>(N*N);
FACTYPE *facp = thrust::raw_pointer_cast(all_fac);

thrust::device_ptr<TOKENTYPE> all_tkn = thrust::device_malloc<TOKENTYPE>(1);
TOKENTYPE *tknp = thrust::raw_pointer_cast(all_tkn);

thrust::device_ptr<float> events = thrust::device_malloc<float>(N*N);
float *ep = thrust::raw_pointer_cast(events);

thrust::device_ptr<int> chzn = thrust::device_malloc<int>(N*N);
int  *cp = thrust::raw_pointer_cast(chzn);

thrust::device_ptr<bool> rdndnt = thrust::device_malloc<bool>(N*N);
bool *rp = thrust::raw_pointer_cast(rdndnt);


...

while (clock < SIMTIME ) {
   thrust::device_ptr<float> mptr =  thrust::min_element(events, events + N*N);

   clock = *mptr + d;

   thrust::device_ptr<int> chzn_last =thrust::copy_if(key,key+N*N,events,chzn,leq(clock));

   int chzn_num = chzn_last - chzn;

   int gridSize = (chozen_num+blocksize-1)/blocksize;

   process_departure<<<gridSize,blocksize>>> (facp,tknp,ep,cp,chzn_num);

   chk_redundant<<<gridSize,blocksize>>> (facp,tknp,ep,cp,rp,chzn_num);

   process_arrival<<<gridSize,blocksize>>> (facp,tknp,ep,cp,rp,chzn_num);
}
```

Fig. 5: The improved implementation using the Thrust library

reasons we use Thrust is that it abstracts away the details of low-level CUDA function calls, such as cudaMalloc, cudaMemcpy, kernel launch, etc. For example, it provides the device pointer which allows programmers access the device memory without calling cudaMemcpy explicitly. The `*mptr` in Figure 5 is such a case. For interoperability with C, the device pointer can be converted into a raw pointer and then the users can use it as a parameter to launch a CUDA C kernel.

Another reason we use the Thrust library is that it provides the `min_element` and the `copy_if` functions. So we don't need to write our own and hence the programming effort can be saved greatly. Furthermore, both functions have been tuned and optimized particularly for the NVIDIA GPGPU architecture. For example, the code used in the old algorithm to find the minimum element based on the parallel reduction method is out-of-date and inefficient. Figure 6 shows the general ideas of how the parallel reduction steps are performed in the old algorithm and in the Thrust library, respectively. The former uses the interleaved addressing approach, in which the distance between the two elements to be compared in the array is doubled for each reduction step. The latter adopts the sequential addressing approach, in which the distance is reduced half in every step. In theory, there is no difference between these two methods because both need $O(\log n)$ steps to find the minimum value among n elements. In practice, the latter is bank conflict free and takes advantage of the CUDA memory coalescing within a warp to improve performance [16].

Another important thing is how to extract the aggregated events from the future event set. It is straightforward that the comparison of each event's timestamp with the interval's upper bound can be done in parallel on each thread. The issue here is the management of the chozen events to run after the comparison. The way how this is implemented is not discussed in [13]. The simplest approach is to let the thread discontinue to run if the selection criteria is not met, while the thread which gets TRUE in the comparison will continue to execute the event, i.e., handling the departure/arrival, updating the facility, generating new events, etc.

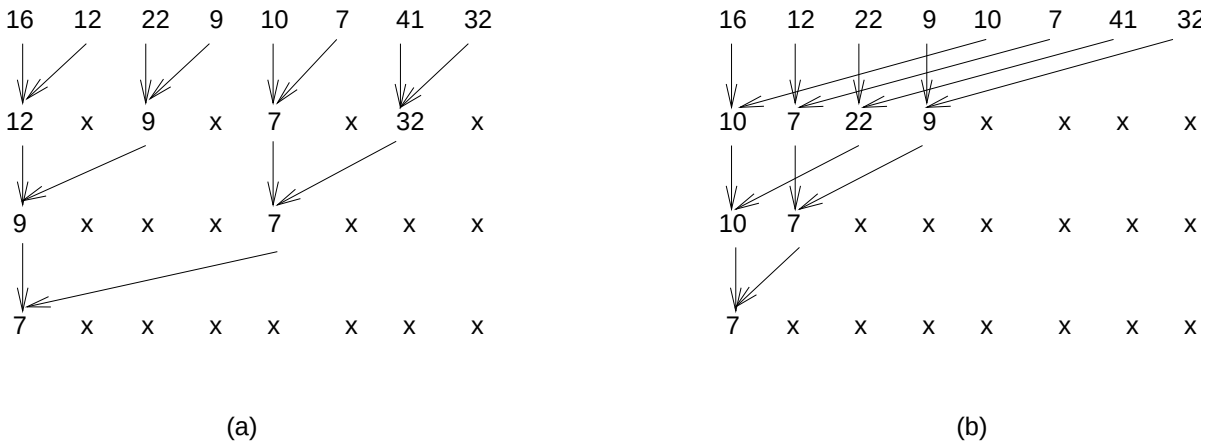


Fig. 6: Parallel Reduction Steps using (a) Interleaved Addressing (b) Sequential Addressing

However, based on our experience, only a small portion of events will be selected in a large-scale simulation. Hence, this approach will cause many threads idle and only two or three threads in a warp can run.

The better approach is to use two phases of processing. In the first phase, the parallel events are collected into an array which stores the identifiers of the selected events. Therefore, the number of the chozen events can be known and then we can run that many of threads to execute the events in the second phase. For collecting the chozen events into an array, each thread needs to figure out the correct position to be stored in the array. There are two implementation methods for this. One method is that we can use an index counter which will be incremented by one for each newly selected event. Since the index counter is shared by many threads, the addition has to be an atomic operation. This can be done by using the CUDA `atomicAdd()` function. Another method, which is used in the Thrust `copy_if` function, adopts the list ranking algorithm with the parallel prefix sum operation[17] to obtain the position of each selected event. Currently, we use the latter because of its availability. However, the choice of which method depends on how many items satisfy the condition. Basically, if the number of items that satisfy the condition is small, using `atomicAdd()` could be better. The empirical comparison of these two methods is worth further investigation.

Figure 7 shows the pseudo code of event execution in the second phase of processing. When a token leaves a facility, the first token, if any, in the waiting will get its service and a departure event will be scheduled for it. For the leaving token, an uniform random variable will be generated to determine its destination and its token identifier and timestamp will be put into the next service facility's corresponding incoming port. Note that it is possible there are more than one token arrived at the same facility. This will cause more than one thread handling of the same facility and mess up the computation. To solve the problem, we use the pre-defined port order, east → south → west → north, to determine which thread has to process the arrivals at the facility. As shown in the Figure 5 and Figure 7, the decision making is a separated kernel launch of the function `chk_redundant()`. For processing the arrivals at a facility, we append all of the incoming tokens to the waiting queue if the service facility is busy. Otherwise, the newly arrived token with the smallest timestamp can start the service, while the rest of incoming tokens will be put in the waiting queue based on their timestamp order.

Note that the processing of the departures, the checking of the redundant threads, and the processing of the arrivals should be launched from the kernel respectively. This is because we have to wait until all of the threads finishes one kernel launch and then start running the next kernel function. Otherwise, the incoming port data will not be consistent due to the clean up at the end of the function `process_arrival()`. Furthermore, the CUDA function `__syncthreads()` cannot be used here as a barrier because it can only synchronize the threads within a warp, not all of the threads.

5. Experimental Results

In this section, we compare our PDES implementation on the GPGPU with a sequential heap-based DES on the CPU. The experimental platform, supported by Ohio Supercomputing Center, has one HP ProLiant SL390s G7 Node with two Intel Xeon x5650 CPUs (2.67GHz, 48GB memory). The OS is 64-bit Linux, kernel version 2.6.32. The GPGPU used in the experiments is a NVIDIA Tesla M2070, which contains 14 multiprocessors (448 CUDA cores in total) and 6GB GDDR5 memory. A warp, the scheduling unit in CUDA, has 32 threads and these 32 threads perform SIMD computation on a multiprocessor. The device programs use CUDA compiler driver 5.0. The parallel algorithm runs on

```
__global__ void process_departure( parameters )
{
   calculate the statistics;
   If the facility's waiting queue is empty
      set the state of the facility to be idle;
   else
      remove the front token from the waiting
         queue and put it in service.;
      schedule a departure event for the token;
   determine destination for the leaving token;
}

__global__ void chk_redundant( parameters)
{
  check the four incoming ports of the facility;
  if there are more than one arrival, use the
     predefined port order to determine which
     arrival's corresponding thread can run.
}

__global__ void process_arrival( parameters )
{
   if the thread is marked as redundant
      return;
   sort the incoming tokens by their timestamps;
   if the state of the facility is idle
      let the first incoming token get the service
           and schedule a departure event for it;
      put the rest of incoming tokens into the
           waiting Q;
   else
      append the incoming tokens to the waiting Q;
   clean up the data in the four incoming ports.
}
```

Fig. 7: Pseudo code for event processing

the host and the device, while the sequential algorithm runs on the host.

The torus queueing network model mentioned in the earlier section was used for the simulation. In the first experiment, we measured the simulation execution times by varying the number of facilities and the interval sizes. The mean service time (i.e. the parameter M in calling the function `expon()`) of the service facility is set to 10. Figure 8 shows the performance improvement in the GPGPU experiments compared to sequential simulation on the CPU. The speedups grow when the number of facilities increases. In particular, our PDES implementation outperforms the sequential DES by 60x speedup for 1024x1024 facilities with $d = 2.0$. The curve also scales well which implies that the speedup could be increased further for simulating a larger scale torus network. It can also be seen in Figure 8 that the larger the interval value d, the larger the speedup obtained. This is because a larger interval allows more parallel events to run. To verify this, we also measured the average number of parallel events for different number of facilities and different interval sizes. The result can be found in Figure 9.

In another experiments, we evaluated the difference in simulation summary statistics due to the use of the modified service time distribution function. Figure 10 shows the difference in the facility server utilization for varied intervals. The simulation with smaller time interval behaves closer to running the simulation with the original service distribution function. As the interval d increases, the utilization also increases because the service time is at least large as d. Figure 11 shows similar effect on the system waiting time, which is the average time of a token staying in a service facility, including the service time and the waiting time in the queue. Unlike utilization, the system waiting time drops as interval increases. For the purpose of comparison, we also used two mean service times: 10 and 20. For the same interval d, the larger mean service time has smaller difference in utilization and system waiting time because the interval d occupies a smaller portion in the service time.

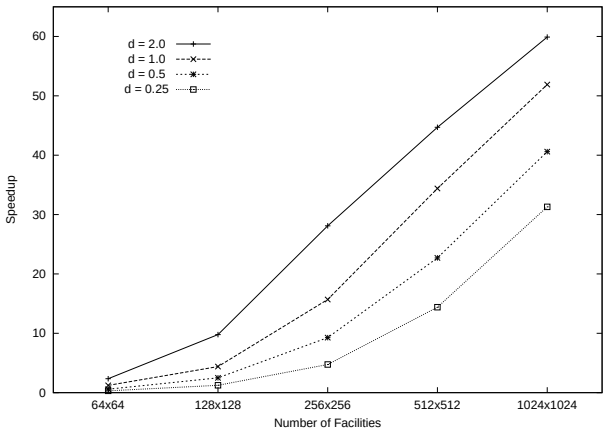


Fig. 8: Speedups

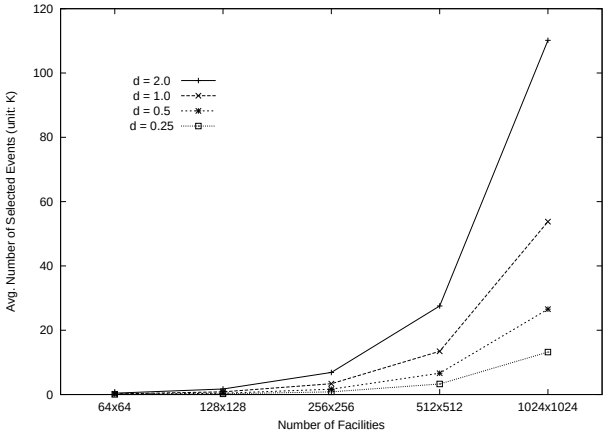


Fig. 9: Average number of parallel events

6. Conclusion and Future Work

We presented a fast implementation of PDES on GPGPU by using the productivity-oriented Thrust library. Our scheme exploits a modified service distribution function to allow clustered events to be processed in parallel, while

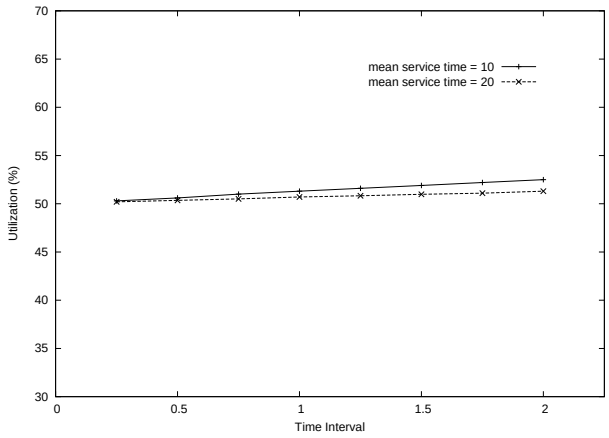


Fig. 10: Utilization with with different time interval d

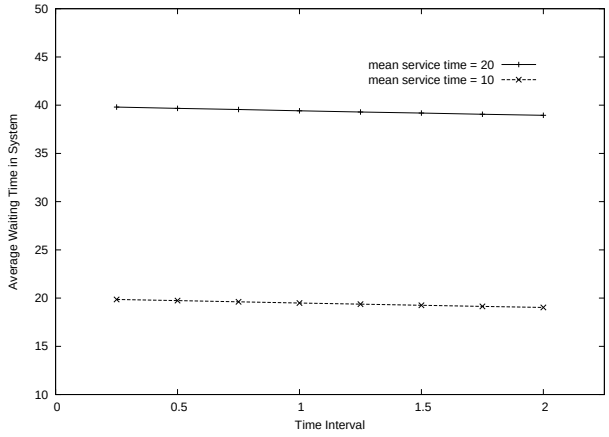


Fig. 11: System Waiting Time with different time interval d

preserving timestamp ordering and causal relationships of events. Thrust, which provides a collection of optimized data parallel primitives such as reduce, stream compaction, prefix sums, etc., makes our implementation more efficient. The experimental results are encouraging. We were able to achieve 60x speedup using our implementation at the expense of accuracy in the results. The speedup curve scales well which indicates that our implementation utilizes the massively data parallel processing power of GPGPU and is suitable for large-scale simulation models.

In the future, we plan to investigate various optimization techniques, such as using shared memory and/or register file, to improve the program performance. Moreover, the performance comparison of Thrust's `copy_if()` and the implementation using `atomicAdd()` in our simulation tool is worth of further studies.

Acknowledgments

This research was supported by a Fulbright Senior Research Grant, facilitated by the Foundation for Scholarly Exchange in Taiwan, and by an allocation of computing time from the Ohio Supercomputer Center.

References

[1] R. Fujimoto, "Parallel discrete event simulation," *CACM*, vol. 33(10), pp. 30–53, 1990.

[2] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Trans. on Software Engineering*, vol. 5, no. 5, pp. 440–452, May 1979.

[3] D. Jefferson, "Virtual time," *ACM Trans. on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[4] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010. [Online]. Available: http://code.google.com/p/thrust/

[5] V. S. Sunderam and V. J. Rego, "Eclipse: A system for High Performance Concurrent Simulation," *Software-Practice and Experience*, vol. 21(11), pp. 1189–1219, 1991.

[6] V. J. Rego and V. S. Sunderam, "Experiments in Concurrent Stochastic Simulation: The Eclipse Paradigm," *Journal of Parallel and Distributed Computing*, vol. 14(1), pp. 66–84, January 1992.

[7] J. West and A. Mullarney, "ModSim: a language for distributed simulation," in *Proceedings of SCS Multiconference on Distributed Simulation*, 1988.

[8] D. Baezner, G. Lomow, and B. W. Unger, "Sim++: The transition to distributed simulation," in *Proceedings of the SCS Multiconference on Distributed Simulatio*, 1990, pp. 211–218.

[9] J. Sang, E. Mascarenhas, and V. Rego, "Mobile-Process Based Parallel Simulation," *Journal of Parallel and Distributed Computing*, February 1996.

[10] E. Mascarenhas, F. Knop, R. Pasquini, and V. Rego, "Minimum cost adaptive synchronization: experiments with the parasol system," *ACM Trans. on Modelingand Computer Simulation*, vol. 8, no. 4, pp. 401–430, 1998.

[11] K. S. Perumalla, "Discrete-event execution alternatives on general purpose graphical processing units (gpgpus)," in *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS '06, Washington, DC, USA, 2006, pp. 74–81.

[12] Z. Xu and R. Bagrodia, "Gpu-accelerated evaluation platform for high fidelity network modeling," in *PADS*, 2007, pp. 131–140.

[13] H. Park and P. A. Fishwick, "A gpu-based application framework supporting fast discrete-event simulation," *Simulation*, vol. 86, no. 10, pp. 613–628, 2010.

[14] H. Park and P. A. Fishwick, "An analysis of queuing network simulation using gpu-based hardware acceleration," *ACM Trans. Model. Comput. Simul.*, vol. 21, no. 3, p. 18, 2011.

[15] R. L. Woods and K. L. Lawrence, *Modeling and simulation of dynamic systems*. Upper Saddle River, NJ: Prentice-Hall, 1997.

[16] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[17] J. C. Wyllie, "The complexity of parallel computations," PhD thesis, Cornell University, Ithaca, NY, USA, Tech. Rep., 1979.