First published: ESM'2016, 30th European Simulation and Modelling Conference, October 26-28, 2016, SIANI - University of Las Palmas, Spain

ARNG: ACCELERATING DISCRETE EVENT SIMULATION RNGs BY ASYNCHRONOUS RANDOM NUMBER GENERATION

Arne Koors and Bernd Page
Department of Informatics
University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
E-mail: {koors, page}@informatik.uni-hamburg.de

KEYWORDS
Discrete Event Simulation, Random Number Generation, Functional Decomposition, Functional Parallelism, PRNG, Parallel Simulation, PDES, DESMO-J.

ABSTRACT

Functional decomposition, also called functional parallelism, was an approach to introduce inner parallelism into discrete event simulators in the 1980s, in order to accelerate simulation experiments. Due to technical restrictions at that time, it did not gain widespread acceptance. This paper introduces modifications of the approach, considering today's technical possibilities. A focus is set on asynchronous execution of simulation infrastructure by thread pools. Applying the concepts of asynchronous functional parallelism to random number generation leads to our proposal for asynchronous random number generation (ARNG), which has been implemented into the discrete event simulation framework DESMO-J. We describe its implementation and report on results of experiments which were conducted to assess the performance potential. The impact of different parameters is analyzed and advice for parameterization is given.
We found that the attainable acceleration factor of our ARNG implementation is limited within a range from circa 2 to 5 on an eight core machine, depending on the concrete distribution and the number of thread pool workers used.
If two (otherwise idling) processor cores are employed, then random number generation can be accelerated by a factor of at least 1.74 in non-trivial simulation models, and by a factor of at least 3.00, utilizing four processor cores.

INTRODUCTION

Discrete event simulation (DES) is a computationally intensive software technique, where event routines with stochastic inter-event times are successively executed, to advance model state in time (Page and Kreutzer 2005).
Nowadays, discrete event simulation users typically employ four to eight core computers for office work; however, their conventional discrete event simulators use rarely more than one processor core. The cause for this disproportionate utilization of processing power is that DES conceptually is a sequential technique: the timely non-coincident event routines are processed one after another, to ensure realistic flow of simulation time from past to future and to guarantee causal validity of experiments.
To overcome this dilemma and better use available computational units (*CU*s, i.e. processor cores, CPUs or computers in a (local) network), historically the following approaches have been discussed, with the objective of accelerating DES experiments by parallelization (fig. 1):

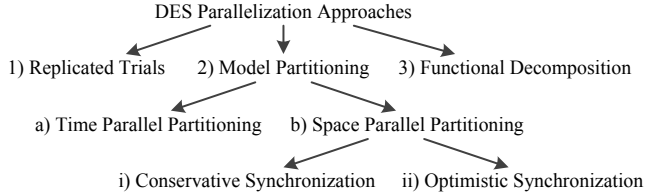

Figure 1: DES Parallelization Approaches

1) For statistical reasons, DES experiments have to be repeated with varying random number generator seeds. These *replicated trials* can be executed in parallel on different CUs; e.g. (Pawlikowski et al. 1994).
2) *Model partitioning* divides simulation experiments into a) disjoint time intervals (*time parallel partitioning*) or b) disjoint sets of state variables (*space parallel partitioning*) (Perumalla 2006, Kunz 2010). Each model partition is assigned to a so called *logical processor* (LP). Simulation experiments are carried out by mapping logical processors to CUs and executing their partitions in parallel.
Whereas time parallel partitioning is only feasible for a limited number of applications (Fujimoto 2000, p.177–191), most research in the last two decades concentrated on space parallel partitioning (cf. Fujimoto 2015): LPs have independent simulation clocks that need to be synchronized to avoid *causal violations*, where events in the present of one LP request modification of state variables in the past of another LP. i) *Conservative synchronization* (Chandy and Misra 1979) ensures that all LPs proceed orderly to prevent causal violations, whereas ii) *Optimistic synchronization* (Jefferson 1985) allows uncoordinated time advance in different LPs, but takes precautions enabling LPs to roll back their local state to previous states, which have been valid before a causal violation occurred.
3) *Functional decomposition*, also called *functional parallelism* suggests to internally parallelize simulators, e.g. by dividing the event list and managing their parts on different CUs (Comfort 1984).

The replicated trials approach 1) is conceptually easy to comprehend and established in practice. Nevertheless, it cannot accelerate individual experiments, as required in online simulations, or in the developmental phase of a simulation model.

Many contemporary publications use the term *Parallel Discrete Event Simulation* (PDES) synonymously with space parallel model partitioning 2b). If a model can be split into seldom communicating sub-models in a straightforward manner, model partitioning is a proven option. However, not all model classes lend themselves for model partitioning equally well. Moreover, having to consider technical constraints of the chosen partitioning and synchronization scheme upstream, in the logical modeling phase, can interfere with reasoning about the original system from an unimpeded functional perspective.

From a historical point of view, work on functional decomposition 3) has mainly been published in the 1980s. The survey (Kaudel 1987) lists ten papers, half of them authored or co-authored by John C. Comfort. Apart from Comfort's primary activities, other authors proposed or announced further work or reported about ongoing research, but most without concrete results. To our best knowledge, we could not find evidence of further work on functional decomposition or functioning implementations in real discrete event simulators after 1993.
The main criticism on functional decomposition is that it does not scale (Fujimoto 2000, p.48). (Kaudel 1987) expects speedup to be "probably limited to a factor of two for most problems". These reservations are caused by the basic determination that simulation infrastructure components like event set processing, random number generation, statistics collection, etc. should be processed by a fixed number of dedicated computers, resp. networked processors.

The remainder of this paper is structured as follows:
The next section modifies the traditional functional decomposition approach with regard to today's technical possibilities; it particularly proposes introducing thread pool based asynchronous parallelism into simulation infrastructure components. The following section describes our implementation of asynchronous random number generation, by applying the principles of asynchronous parallelism to parallel random number generation. Subsequently, we report on results of experiments that were conducted to assess the performance potential of asynchronous random number generation. Last, we give an outlook and a conclusion.

GENERAL APPROACH

Nowadays, office and even laptop computers have a suitable number of cores and run under multi-threaded operating systems, providing access to shared memory and lightweight synchronization primitives by comfortable APIs.
Against this background, we were interested to explore whether the original idea of accelerating DES by functional parallelism could successfully be adapted to today's technical possibilities. The original concept was modified on some points:

1. We do not employ networked computers for simulation infrastructure anymore, but focus on processor cores within single processors, with access to shared memory.
2. No fix assignment of simulation infrastructure components to CUs is made. Instead, method invocations of simulator sub-components are flexibly executed on demand, by a thread pool which itself is backed by the processor cores.
3. For performance reasons, the simulation infrastructure may internally communicate by shared memory concurrency.
4. Special attention is directed to asynchronous parallelism as a means of potential experiment acceleration.

Items 2 and 4 of the list above are detailed in the following.

Thread Pools for Simulation Infrastructure Execution

It is advisable to distribute simulator functionality like event set processing, random number generation, waiting queue operation, etc. to respective simulation infrastructure components. There may be more than one object per component type (e.g. several random number generators), and components should be structured into fine-grained sub-components (fig. 2, left side). Sub-components are passive objects without an own thread. For execution of simulator functionality, references to methods of sub-components are committed to a thread pool, which consists of several workers (fig. 2, center). Every thread pool worker has an own thread and can execute one sub-component method at a time. In this way, sub-components gain processing power only when needed, borrowing it from thread pool workers temporarily, and on demand.
The operating system assigns thread pool workers to physical processor cores (fig. 2, right side). Depending on the number *TPW* of thread pool workers and the number of actually available processor cores *PC*, several simulator functionalities can be executed in parallel.

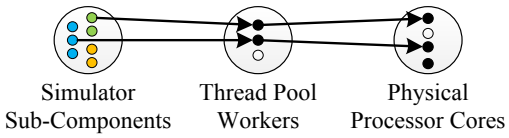

Figure 2: Relations between Simulator Sub-components, Thread Pool Workers and Processor Cores

Figure 2 shows a situation where two simulator sub-component methods are executed by two thread pool workers in parallel. One of the three thread pool workers is idle at the moment, one processor core is idle, and one core is executing other non-simulation functionality.
If more method calls are scheduled into the thread pool than available workers exist, then the calls will be implicitly queued. Generally, multi-tasking within the thread pool is cooperative, not preemptive. Therefore, sub-component methods should be short and non-blocking, to avoid degeneration of simulator responsiveness.
By contrast, thread pool workers will be executed by processor cores in a preemptive way in most modern operating systems. Hence, *TPW* may be set to $TPW > PC$, if long and/or blocking method calls in the simulation infrastructure cannot be avoided. In this way, long sub-component methods will be interrupted indirectly, because the backing cores will be re-assigned to other thread pool workers by the operating system.
The finer grained the simulator sub-component functionality is, the better the simulator will scale when *TPW* is increased

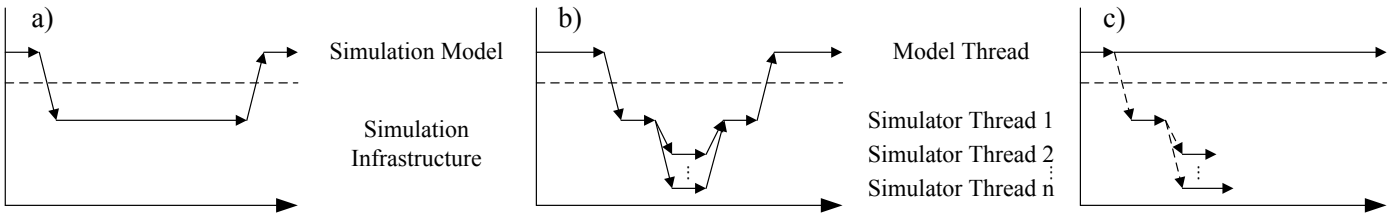

Figure 3: a) Sequential Execution, b) Synchronous Parallelism, c) Asynchronous Parallelism in Simulation Experiments

(e.g. due to subsequent processor generations). Therefore, it is desirable to subdivide simulation functionality into reasonably fine-grained sub-components from the beginning.

Asynchronous Parallelism in Simulation Infrastructure

In conventional discrete event simulators, one thread sequentially executes model logic as well as simulator functionality (fig. 3a): all method calls to simulation infrastructure are executed by the same (model) thread.
Synchronous parallelism tries to accelerate simulation functionality by distributing it to parallel threads in a fork-join way (fig. 3b): The model thread calls a separate simulator thread, which in turn delegates sub-tasks to parallel subordinated worker threads. When all workers have finished, the individual results are consolidated, and the consolidated result is returned to the waiting model thread. Dependent on the specific task and the number of workers employed, faster simulator response may be obtainable. In consequence, proportionally more time may be spent for model computation, and the simulation experiment may terminate earlier. However, the model thread has to wait for the simulator thread(s), at least for as long as the longest internal sub-method call takes, plus the additional fork-join overhead.

We propose applying *asynchronous parallelism* to discrete event simulation infrastructure. Model thread and simulator infrastructure components should be decoupled to the greatest possible extent.
In this case, the model thread can invoke simulator functionality asynchronously, meaning that the model directly proceeds with its own functionality after invocation (fig. 3c): in many use cases, the model does not necessarily have to wait for termination of the called simulator functionality. Moreover, simulator sub-components should perform asynchronously as well, parallel on several thread pool worker threads.
Compared to sequential or synchronous parallel execution, the model thread is not burdened or blocked with regard to simulator functionality but can immediately proceed with model logic, while simulator functionalities execute asynchronously and in parallel in the background.

Naturally, all *downstream* simulator functionality like statistics, writing of trace or log files and display of run time status information resp. visualization offers itself for asynchronous parallelism. Nonetheless, *upstream* components like random number generation can be implemented in an asynchronous manner as well, as this paper demonstrates. Further, we are convinced that *midstream* components like event scheduling and waiting queue management are able to operate asynchronously in parallel, too. Generally, designing simulation infrastructure components with asynchrony in mind will enable simulation experiments to progress faster.

The revised and updated form of asynchronous functional parallelism delineated above has notable potential for acceleration of traditional sequential DES experiments. Besides, its concept does not oppose the established parallelization approaches (model partitioning and replicated trials), but can orthogonally complement them as well, by putting otherwise unused processor cores to work.

RANDOM NUMBER GENERATION

This section first gives a short overview of parallel random number generation and then presents our implementation of asynchronous random number generation in DESMO-J. DESMO-J (www.desmo-j.de) is our open source discrete event simulation framework in Java, developed and maintained by the modeling and simulation workgroup at the University of Hamburg (Göbel et al. 2013).

Parallel Random Number Generation

Modeling systems in discrete-event style very often involves mapping parts of the real system to stochastic distributions, because a) the original system actually behaves randomly, b) inter-relations beyond the system boundary are modeled stochastically (e.g. inter-arrival times) or c) potentially deterministic internal system details of secondary relevance are idealized to stochastic processes (e.g. service times).
Algorithmic random number generators (RNGs) provide stochastically distributed number streams. They often base on uniformly distributed random numbers; more complex distributions are obtained by applying a transformation function or another algorithm to these uniformly distributed random numbers (e.g. Page and Kreutzer 2005, pp.161–168).
Essential requirements for algorithmic random number generation are a) statistical soundness and b) reproducibility.
a) Statistical soundness means that random numbers *within one random number stream* are statistically independent; no predictable patterns should be produced and the generated numbers should be uncorrelated and non-cyclic; or at least cycles have to be very, very long (Srinivasan et al. 1999).
b) Reproducibility of random numbers is important to make stochastic experiments repeatable, e.g. for debugging, analysis of model parameter variation, or for verification by other scientists.
Intensive research has gone into high quality *sequential* random number generation; a good overview is given in (Knuth 1997, chap. 3). For the purpose of this paper, it is

enough to be sure that very good off-the-shelf sequential random number generators exist. For example, our simulation framework DESMO-J uses a Mersenne Twister or alternatively a Linear Congruential Generator.

Parallel random number generators (PRNGs) are concerned with providing reproducible streams of random numbers that are statistically sound *also* when examining *pairs or sets of random number streams*. Moreover, PRNGs should be scalable and without data sharing or synchronization apart from an initialization phase (Coddington 1997; Srinivasan et al. 1999). The techniques employed here either partition one sequential random number stream into different sub-streams returned to parallel client threads (*sequence splitting*, *random spacing* or *leap frog*) or generate different random number streams per client, by individually parameterizing algorithms.
In the context of this paper, it should be stressed that parallel random number generation is *not* about accelerating random number production by *n* threads working in parallel on *one* random number stream. PRNG rather ensures that *n* threads concurrently producing *n* random number streams (with no acceleration at all) will maintain statistical soundness, when pairs of these random number streams are examined.
A good survey on parallel random number generation is given in (Hill et al. 2013). For the purpose of this paper, it is sufficient to be aware that decent parallel random number generators are available (e.g. JAPARA (Coddington and Newell 2004) or SPRNG (Mascagni et al. 1999)). Any of these PRNGs can be used as a high-quality black-box component in asynchronous random number generation, as exposed in the following sub-section.

Asynchronous Random Number Generation

The main idea of asynchronous random number generation (ARNG) is to produce blocks of random numbers in advance, in order to immediately provide single random numbers when a distribution is sampled.
A similar concept is described in (Hill 2003): Here, millions of random numbers are pre-computed before the start of an experiment. The generated stream of random numbers is pre-compiled and linked to the simulation program, like a program library. Whenever the simulation model needs to sample random numbers, the pre-computed stream is unrolled and single numbers are drawn from it, with short constant access time.
Our asynchronous functional parallelism approach provides short constant access times as well, but differs in that it neither needs a separate pre-computation phase nor memory for millions of random numbers. Instead, the stream of random numbers is pre-computed on demand, parallel to a simulation experiment, and in an asynchronous and block wise manner.

Our implementation presented here is only one of several variants conceivable. More sophisticated procedures may be introduced in the future.
The asynchronous random number generator described here consists of two types of sub-components:

A *random number cache* (hereinafter referred to as *cache*) holds a reference to a block of pre-computed random numbers (cf. fig. 4, left). If a sample is requested, the next random number from this *current sample block* is read and instantly returned.
If the whole current sample block is consumed, a new block is taken from one of the internal *cache lines* of the cache. Cache lines are used cyclically, in a round robin procedure.
A variable called *current cache index* refers to the current cache line. For replacement of the exhausted block, the current cache index is set to the following cache line. Then, the thread utilizing the cache (usually the model thread) tries to copy the sample block reference of the (newly) indexed current cache line to the current sample block variable. If this succeeds, the content of the current cache line is invalidated and a corresponding random number producer sub-component is scheduled into the thread worker pool, for generation of a new sample block. Finally, the first sample of the (new) current sample block is returned to the model.

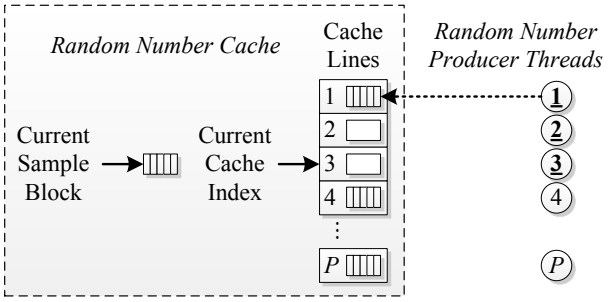

Figure 4: Sub-Components of Asynchronous Random Number Generation

If the cache cannot copy the content of the new current cache line because of invalidity, then the executing (model) thread blocks until the awaited sample block will be inserted into the current cache line.

Random number producers (hereinafter called *producers*) are sub-components which access basic RNG or PRNG functionality (see previous sub-section). On execution by a thread pool worker, a producer creates an array of *block size* samples and inserts a reference to this array into its corresponding cache line (see fig. 4, right, producer no. 1).
If the cache was blocked awaiting just this cache line, the model thread will be resumed, move the new sample block reference to the current sample block variable, invalidate the cache line again and instantly re-order another sample block by re-scheduling the producer.
If the cache was not blocked awaiting the inserted cache line, it will simply continue its normal operation (cf. fig. 4).
Cache lines should be implemented using a synchronized data structure like a Java `ArrayBlockingQueue` of size 1 (i.e. space for 1 sample block reference per cache line).
By the execution sequence given above, producers are only scheduled if their corresponding cache lines are empty. Therefore, no producer can insert more than one sample block into its cache line. Moreover, each producer holds only one private reference to its corresponding cache line. It is impossible for a producer to access cache lines of other producers.
Upon cache initialization, producers are immediately scheduled for asynchronous pre-production of sample blocks

to cache lines. Thus, samples may already be available on first demand of a model thread.

Fig. 4 shows a situation where the model thread plus three thread pool workers execute in parallel: The model thread accesses the current sample block of the random number cache (originally stemming from cache line 3), while producer 1 inserts his pre-computed sample block into cache line 1, and producers 2 and 3 are still busy with re-generating their sample blocks.

It should be noted that the random number stream created by this implementation is reproducible, but may differ from a sequentially generated random number stream, because here sample blocks originating from *different* (identically distributed) random number streams are deterministically interlocked.
If no basic PRNG is used, then statistical soundness of the resulting stream output by the cache has to be investigated carefully: Though all sample blocks trace back to the same distribution, an unfortunate combination of long-distance correlation in the underlying random number stream together with producers starting their block sequences at just these correlated entry points may lead to short-distance correlations in the random number stream provided by the cache. We apply random seeding to mitigate this problem, in case producers use conventional RNGs like the Mersenne Twister.
However, if an advanced PRNG is used which supports the so-called *jump-ahead* technique (random numbers *s* steps ahead in the stream are directly accessible), then *p* producers can use *p* parallel entry points with *block size* distance into the same underlying random number stream. In this case, the cache will reproducibly output the same sound random numbers as the underlying sequential stream, but more rapidly, because obtained in parallel.

Two further options should be mentioned:
First, the cache may adapt the size of sample blocks requested from producers dynamically (`blockSize` is a parameter of the producer's `requestSampleBlock` method). If the model needs samples at higher frequencies, then smaller block sizes are advisable: they allow thread pool workers to switch more quickly between different sub-components, thereby increasing responsiveness of the thread pool as a whole.
As a second option, the cache may create or lay off producers dynamically, for similar motives as above.

EXPERIMENTAL RESULTS

This section compares performance impacts of asynchronous random number generation with conventional sequential random number generation.

Experiment Design

All experiments have been carried out on a Windows 7 64 bit machine with 8GB RAM and an Intel i7-3610QM processor, running at 2.30 GHz with 4 physical cores plus 4 hyper-threaded cores.

Overall, 650 random number generation experiments have been performed, each based on 100 replications of sampling 10 million random numbers by the model thread.

The following parameters were varied per experiment:
– Distribution type (Uniform, Exponential, Normal and Poisson; Poisson distribution with different *mean* parameter settings); or mixes of the aforementioned distribution types
– Number of thread pool workers (1 up to number of processor cores, i.e. 1…8, cf. fig. 2). The number of cache lines equaled the number of processor cores, i.e. 8, cf. fig. 4.
– Sample block size (4^3…4^{10} samples per sample block, i.e. 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576)

To compensate for runtime irregularities like garbage collection etc., 101 replications were performed per experiment. Results of the first replication were discarded; it was considered a warm up replication. The time needed for consecutively sampling 10 million random numbers was measured and arithmetically averaged for the following 100 replications. A pause of 3 seconds was applied between replications, as cool-down period.
Experiment setup time and other non-sampling times were not incorporated in the experimental results, but pre-production of producers upon cache initialization was included.
Apart from 10 sequential base line experiments, production and consumption of random numbers occurred asynchronously and concurrently in all other experiments.
In consequence, every data point in the following line graphs refers to 1 random number generation experiment backed by 100 replications of sampling 10 million random numbers.

All experiments were implemented in Java 8u101, using our discrete event simulation framework DESMO-J. Every experiment consisted of initializing the simulation infrastructure and scheduling one single external event which then sampled ten million random numbers in a for-loop.
The basic uniform random number generator used was the same Mersenne Twister as in the standard DESMO-J sequential random number generators.
The algorithms that transformed uniform random numbers to different distributions were the same as those used for sequential random number generation. In fact, the standard algorithms were passed as constructor parameters (Lambdas in Java 8) to random number producers.
Different instances of producers were initialized by random spacing (random seeding).

Additional Memory Consumption

Additional storage requirements of asynchronous random number generation are determined by a) the number of cache lines *CL* per cache and b) the sample block sizes *SBS* (cf. fig. 4).
In total, (*CL* + 1) * *SBS* * 8 Bytes of memory are needed per cache, e.g. 72MB on our machine with 8 cache lines + 1 current sample block, and sample block size 1,048,576. 8 Bytes are necessary for storage of the Java data type *long* resp. *double*.
Generic implementations will have a higher memory footprint, because in this case the sample blocks hold

references to *Long* or *Double* objects. On 64 bit machines, these references have a size of 8 Byte as well, but now additional 8 Byte per *Long* or *Double* object referred to will double the storage requirements.
Though maximally 144MB seem to be tolerable on today's computers, later results will show that very high sample block sizes should be avoided anyway, since they increase sampling times and congest the thread pool.

Effort Ratios of Sequential Random Number Generation

The time necessary for random number generation varies, depending on the concrete type of distribution chosen. When time for sequential generation of ten million uniformly distributed random numbers is set as 100%, then sequential generation of exponentially distributed random numbers takes about 60% longer in DESMO-J (cf. fig. 5), and sequentially generating normally distributed samples takes about twice the time of uniformly distributed random numbers.

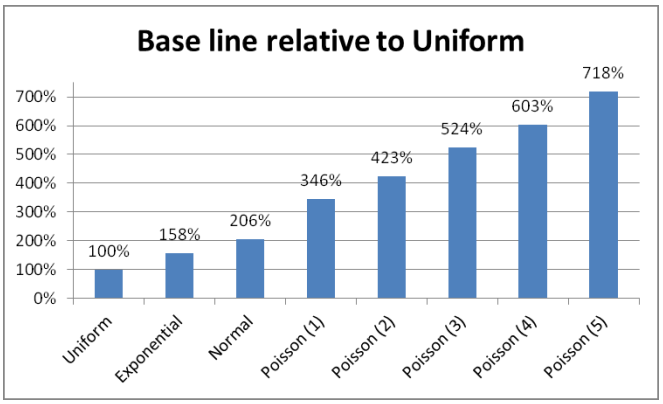

Figure 5: Ratios of Sequential Random Number Production Times

These relationships hold independently of the respective distribution parameters (lower and upper bound; mean; standard deviation). However, generating samples of Poisson distribution *is* dependent on the parameter value (mean) in DESMO-J. It will take about 345% to 720% of the time needed for uniform random number generation, if the mean is in the range from 1.0 to 5.0, and longer for higher parameter settings. We will further refine our algorithms, but for the sake of this paper, the different Poisson generation efforts will be regarded as instances of computationally intensive random number generation algorithms of any distribution.

In the following, the sampling times for conventional sequential random number distributions are set as *base lines* for comparison with their asynchronous variants. The *Acceleration Factor AF* is defined as the quotient of the base line and the sampling time for a specific combination of thread pool size and sample block size, per distribution.
$AF = 1$ means that conventional random number generation and asynchronous random number generation need the same time, whereas $AF = 2$ means that asynchronous random number generation only takes half the time of the base line, resp. samples can be drawn at twice the speed of conventional random number generation.

Impact of Sample Block Size

First, the impact of sample block size on the acceleration factor is studied. Fig. 6 visualizes experimental results for asynchronous random number generation for a) one Uniform distribution b) one Poisson(5) distribution and c) a distribution mix, mimicking a minimal client-server system with one Exponential, one Uniform and two Normal distributions, for inter-arrival times, job type choice and service times of two servers.
Different sample block sizes are mapped to the line graph abscissas, the acceleration factor AF to the ordinates, and 8 lines with different colors and icons denote thread pool sizes from 1 to 8 (black to red) thread pool workers (*TPW*).
Please note that line graphs ordinates have differing scales.

Obviously, extreme sample block sizes (*SBS*) have a negative influence on the acceleration factor AF: Irrespective of distribution and number of thread pool workers, all lines rise for *SBS* < 1,024 and almost all fall for *SBS* > 65,536. By contrast, the AF stays on a central plateau for *SBS* in [1024; 65536], with different variations per distribution (mix). Graphs for other distributions or distribution mixes (not shown here) confirm this finding.

Accordingly, it is advisable to generate sample blocks with sizes in the interval [1024; 65536] for best asynchronous random number generation performance in practice.
We recommend smaller sample block sizes, because then thread pool workers are released more quickly and can turn to serving other requests faster. In consequence, the simulation background architecture becomes more responsive, which finally helps the foreground simulation model thread to proceed quicker.

To facilitate further experimental result visualization, sample block size is set w.l.o.g. to 4,096. In consequence, additional

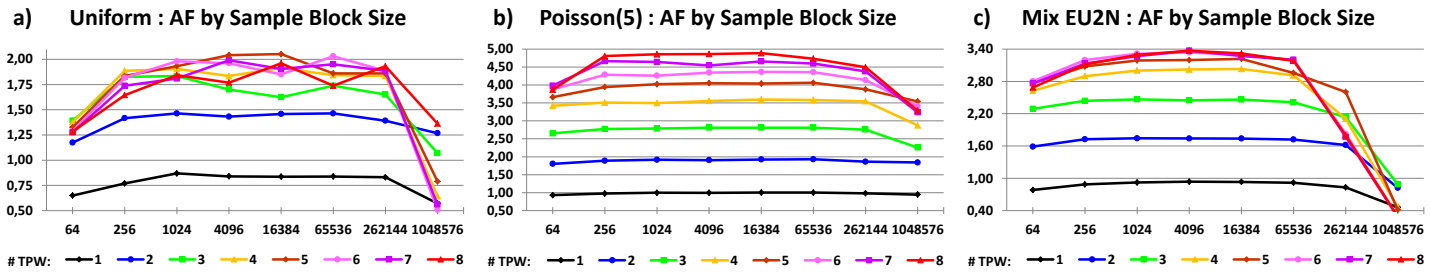

Figure 6: Acceleration Factor by Sample Block Size for a) Uniform Distribution, b) Poisson(5) Distribution, c) Distribution Mix

memory demand for 8 + 1 sample blocks is reduced to 576KB per distribution (in generic implementations).

Impact of Thread Pool Size

Figure 7 relates the number of workers employed in the thread pool (1…8) to the obtainable acceleration factor AF for different distributions and distribution mixes. As stated above, all data points in this graph base on sample block size *SBS* 4,096, but graphs for other *SBS* closely resemble figure 7 when *SBS* is in the practically advisable range from 1,024 to 65,536.
The seven acceleration lines base on the following distributions: Uniform (black, U), Normal (blue, N), Exponential (green, E), Mix 1 (orange, M1), Mix 2 (brown, M2), Poisson(1) (purple, P(1)) and Poisson(5) (red, P(5)).
Mix 1 consists of 1 Exponential, 1 Uniform and 2 Normal distributions, sampled consecutively (cf. fig. 6c).
Mix 2 mimics a more comprehensive client-server scenario with 5 Exponential, 5 Uniform and 10 Normal distributions, modeling different inter-arrival times, job type choices and service times.

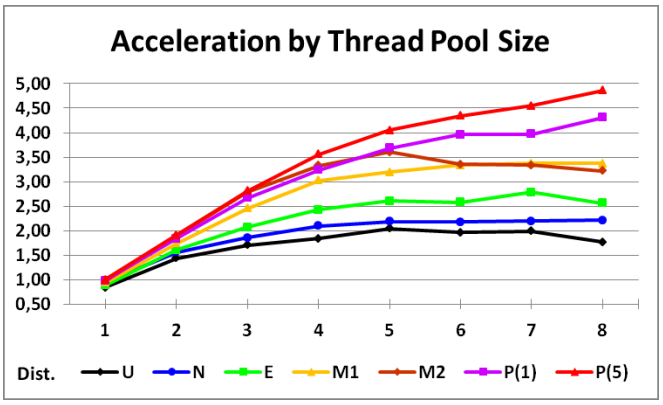

Figure 7: Acceleration Factor by Thread Pool Size for different Distributions and Distribution Mixes

For one thread pool worker, the acceleration factor AF ranges from 0.84 (Uniform) to 1.0 (Poisson(5)), indicating that asynchronous random number generation comes with a certain overhead. Naturally, the overhead is relatively higher for fast Uniform random number generation (16%) and almost negligible for the computationally intensive Poisson(5) distribution (< 0.5%), cf. fig. 5.
The three basic Uniform, Normal and Exponential distributions benefit from asynchronous random number generation with AF ranging from 1.43 (Uniform, 2 workers) to 2.79 (Exponential, 7 workers). These lines reach saturation plateaus when more than five workers are employed. Five workers obviously suffice for asynchronous pre-computation of random numbers here; there is no additional utility in bigger thread pool sizes.

Practically relevant simulation models typically will use more than one distribution. It is noteworthy that the mixes M1 and M2 of the basic three distributions benefit more than the basic distributions themselves from asynchronous random number generation. This can be explained by reasoning that consecutive cyclic sampling decreases the sampling frequency per distribution, while at the same time more background workers can be used for parallel cache line re-generation. In consequence, we measured an AF from 1.74 (Mix 1, 2 workers) up to 3.60 (Mix 2, 5 workers). The Mix AFs surpassed the basic distribution AFs by up to 81% (Mix 2 AF 3.32 vs. Uniform AF 1.83, both with 4 workers).

The more computationally intensive random number generation for a distribution or a mix of distributions is, the better asynchronous random number generation scales with increasing thread pool size. As opposed to lines for Uniform, Normal and Exponential distributions, which remain on horizontal plateaus for more than five workers, the lines for the Poisson distributions continuously slope upwards, reaching an AF of 4.86 (Poisson(5)) with eight thread pool workers in use.
It is comprehensible that comparatively more time spent in random number generation methods (cf. fig. 5) can be better offset by more parallel worker threads than fast random number generation like in the Uniform distribution case: Longer RNG method calls have a better ratio of additional acceleration versus overhead for asynchronous operation; thus more workers can gradually be employed, until the specified ratio approaches 1.0 and the acceleration line finally reaches saturation.

Summarizing fig. 7, the acceleration factor rises when more thread pool workers are employed for asynchronous random number generation. However, the attainable maximum acceleration factor seems to be limited to a range from around 2.0 (single basic distributions) to around 3.5 (mixes of basic distributions). These acceleration factors are reached when employing 5 to 6 workers; therefore asynchronous random number generation does not scale unlimitedly with thread pool size. Only some computationally intensive distributions like the DESMO-J Poisson implementation benefit from further workers and can approach acceleration factors of 5.0 and possibly more.
Non-trivial simulation models with a mix of random number generators similar to Mix 1, or more comprehensive ones like Mix 2, experience an acceleration factor of at least 1.74, if two workers are employed, and an acceleration factor of at least 3.02 when utilizing four workers.

In consequence, it seems worthwhile to employ two to four otherwise idling processor cores for asynchronous random number generation, by accordingly dimensioning the thread pool. However, it should be noted that the operating system finally assigns processor cores to thread pool workers. Therefore it is not guaranteed that every worker is continuously backed by a processor core. Vice versa, processor cores are not tied to workers, meaning that the physical cores may be used for other tasks (e.g. other simulation functionality) when workers in the thread pool idle.

OUTLOOK

The experiments reported about are artificial in that they only sample random numbers, but do not carry out other simulation or model functionality. This is of course necessary and justified when studying asynchronous random number generation on its own, uninfluenced by side effects of other program code. But it would of course be interesting

to examine the interdependencies when utilized in realistic simulation models and – even more – when applied together with other simulation infrastructure components that implement the concepts of asynchronous functional parallelism as well.
For example, we expect that in conjunction with real simulation code, fewer parallel workers will suffice for the same acceleration factors as stated in the section above, because of reduced utilization intensity of asynchronous random number generation and thus a lower demand frequency of sample block re-generation.

Apart from studying asynchronous random number generation "in the wild", we intend to introduce the principles of asynchronous functional parallelism into further components of DESMO-J, like event set processing, waiting queue operations and statistics, to accelerate standard sequential simulation experiments in a transparent way.

CONCLUSION

This paper was concerned with conception and implementation of asynchronous random number generators, applying the principles of asynchronous functional parallelism to random number generation.
The experiments conducted gave an overview of the potential and limitations of our implementation of asynchronous random number generation in discrete event simulators and discussed settings of parameters.
Specifically, it is recommended to prefer smaller sample block sizes of the interval [1024; 65536], to minimize additional memory consumption and increase thread pool responsiveness, with regard to the main simulation model thread.
It is particularly worthwhile using two to four otherwise idling processor cores, backing a simulation worker thread pool of equal size. For non-trivial simulation models using four distributions or more, an acceleration factor of about 1.75 for two workers and of at least 3.0 for four workers can be expected, compared to conventional random number generation.
Though highly computationally intensive distributions can benefit from further workers and approach an acceleration factor of 5.0 on an eight core machine, acceleration of asynchronous random number generation does not scale linearly up to an arbitrary number of processor cores.
However, if otherwise idling cores are available, there is no reason to not apply ARNG, as proposed in this paper.

REFERENCES

Chandy, K.M. and J. Misra. 1979. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs". *IEEE Transactions on Software Engineering,* Number SE-5, No.5, 440–452.

Coddington, P.D. and A.J. Newell. 2004. "JAPARA - A Java Parallel Random Number Generator Library for High-Performance Computing". In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04),* Number 5, 156–166.

Coddington, P.D. 1997. "Random Number Generators for Parallel Computers". *Northeast Parallel Architecture Center*.

Comfort, J.C. 1984. "The simulation of a master-slave event set processor". *Simulation,* Number 42, No.3, 117–124.

Fujimoto, R.M. 2000. *Parallel and Distributed Simulation Systems.* Wiley, New York.

Fujimoto, R.M. 2015. "Parallel and Distributed Simulation". In *Proceedings of the 2015 Winter Simulation Conference,* 45–59.

Göbel, J.; P. Joschko; A. Koors; and B. Page. 2013. "The Discrete Event Simulation Framework DESMO-J: Review, Comparison to other Frameworks and Latest Development". In *Proceedings of the 27th European Conference on Modelling and Simulation,* European Council for Modelling and Simulation, W. Rekdalsbakken, R.T. Bye; and H. Zhang (Eds.) (Aalesund - Norway, 27th-30th May 2013), 100–109.

Hill, D.R.C. 2003. "URNG: A portable optimization technique for software applications requiring pseudo-random numbers". *Simulation Modelling Practice and Theory,* Number 11, No.7, 643–654.

Hill, D.R.C.; C. Mazel; J. Passerat-Palmbach; and M.K. Traoré. 2013. "Distribution of Random Streams for Simulation Practitioners". *Concurrency and Computation: Practice and Experience,* Number 25, No.10, 1427–1442.

Jefferson, David, R. 1985. "Virtual time". *ACM Transactions on Programming Languages and Systems,* Number 7, No.3, 404–425.

Kaudel, F.J. 1987. "A literature survey on distributed discrete event simulation". *ACM SIGSIM simulation digest,* Number 18, No.2, 11–21.

Knuth, D.E. 1997. *The Art of Computer Programming - Seminumerical algorithms.* Addison-Wesley, Reading, MA.

Kunz, G. 2010. "Parallel Discrete Event Simulation". In *Modeling and Tools for Network Simulation.* Springer, 121–131.

Mascagni, M.; D. Ceperley; and A. Srinivasan. 1999. "SPRNG: A Scalable Library for Pseudorandom Number Generation". In *Proceedings of the 9th SIAM conference on parallel processing for scientific computing*.

Page, B. and W. Kreutzer. 2005. *The Java simulation handbook. Simulating discrete event systems with UML and Java.* Shaker, Aachen.

Pawlikowski, K.; V.W.C. Yau; and D. McNickle. 1994. "Distributed Stochastic Discrete-Event Simulation in Parallel Time Streams". In *Proceedings of the 26th Winter Simulation Conference,* 723–730.

Perumalla, K. S. 2006. "Parallel and Distributed Simulation: Traditional Techniques and Recent Advances". In *Proceedings of the 38th Winter Simulation Conference,* 84–95.

Srinivasan, A.; D.M. Ceperley; and M. Mascagni. 1999. "Random Number Generators for Parallel Applications". *Advances in chemical physics,* Number 105, 13–36.

AUTHOR BIOGRAPHIES

ARNE KOORS obtained his master degree in Computer Science from University of Hamburg, Germany. Since then he has been working as a software developer and management consultant in the manufacturing industry, primarily in the field of demand forecasting and planning. Furthermore, he works as a research associate and on his PhD thesis in the Modelling & Simulation research group led by Prof. Dr. Page.

BERND PAGE holds degrees in Applied Computer Science from the Technical University of Berlin, Germany and from Stanford University, USA. As professor for Modelling & Simulation at the University of Hamburg he researches and teaches in Computer Simulation and Environmental Informatics. He is the head of the workgroup which developed DESMO-J and the author of several simulation books.