A parallel implementation of spatially distributed stochastic chemical kinetics

Pontus Melin
Abstract

A parallel implementation of spatially distributed stochastic chemical kinetics

Pontus Melin

Stochastic simulation of reaction kinetics has emerged as an important computational tool in molecular systems biology and will likely continue to grow in importance as experimental techniques are further developed and spatial models can be calibrated to biological data. Many applications require a large number of sample realizations to be generated in order to allow for useful statistical analysis. Exploring different parameters or estimating responses to stimuli adds further complexity such that the generation of tens of thousands of independent realizations is not uncommon. For these applications computational efficiency is an important concern. This study concerns a replica parallel implementation of a stochastic simulation algorithm with the aim of increasing simulation efficiency and explores related concepts such as random number generation in the context of multiprocessing. A handful of different random number generators were profiled in a multiprocessing context. This led to the identification of performance issues with the drand48 random number generation algorithm and allowed for careful selection of a more appropriate random number generator to be used with the parallel implementation. Two alternative solutions were implemented and integrated into the URDME framework. Results showed that the parallel implementations reached speedups up to around 3x on a 12 core machine compared to the serial solution when simulating multiple replicas.
Contents

1 Introduction 6
  1.1 Related work ........................................... 7

2 Background 8
  2.1 Unstructured Reaction-Diffusion Master Equation (URDME) ... 8
     2.1.1 Solvers ............................................. 9
     2.1.2 Stochastic Simulation Algorithm (SSA) ............ 9
  2.2 Random number generators ................................ 10
     2.2.1 drand48 and cache contention ....................... 10
     2.2.2 rand .............................................. 11
     2.2.3 GNU Scientific Library (GSL) ....................... 11
  2.3 Multiprocessing ........................................... 11
     2.3.1 Open Multi Processing (OpenMP) .................... 12
     2.3.2 Expected performance .............................. 12

3 Proposed Solution 13
  3.1 Random number generation ................................ 13
  3.2 Multiprocessing .......................................... 13
     3.2.1 Task based solver .................................. 14
     3.2.2 Transparent solver .................................. 14

4 Implementation 15
  4.1 SSA solver ............................................... 15
     4.1.1 Task based solver .................................. 15
     4.1.2 Transparent solver .................................. 16
  4.2 RNG interface ............................................. 17
  4.3 URDME extensions ........................................ 18

5 Evaluation 20
  5.1 Simulation ................................................ 20
  5.2 Multiprocessing RNG performance ......................... 21
  5.3 Serial vs. parallel ....................................... 22

6 Discussion 25

7 Conclusions and future work 27

8 References 28
1 Introduction

Historically, CPU hardware developments have led to an exponential increase in circuit complexity[1] with part of it being attributed to the decreasing size of transistors. Along with the increasing circuit complexity the clock frequency has increased allowing faster computation. Modern developments however, seems to point towards a future where scaling down the transistors further will prove more challenging as physical limitations kick in[2]. In order to gain significant performance benefits hardware manufacturers have now also turned towards increasing the amount of cores within the processors. Subsequently, software based techniques for distributing workload across the cores have been developed. Such techniques can have large performance benefits for applications that allow for concurrent execution.

Performing statistical analysis of a simulated system often requires a large magnitude of realizations for the analysis to be meaningful and as such computational efficiency is an important concern for these simulations.

In this study the open source API Open Multi-Processing (OpenMP) is used to implement a parallel algorithm for simulating chemical reactions within a volume using Gillespie’s direct algorithm. The aim is to provide an efficient algorithm which exploits independencies in the problem model. Multiprocessing principles are applied to Gillespie’s direct algorithm which computes reaction rates independently across multiple sub-volumes and updates the molecule counts accordingly[3].

Spatial coupling could be introduced by executing a spatial coupling algorithm on the product of Gillespie’s direct algorithm after some amount of time. Seeing as spatial coupling would be more difficult to parallelize as data dependencies increase significantly there is an interest in executing the non spatially-coupled Gillespie’s direct algorithm efficiently.

The workload of independent instances is assigned to different cores and thus a decrease in total execution time is expected when executing on a multi-core machine. Gillespie’s direct method is a variant of a dynamic Monte Carlo method and as such relies on the generation of random numbers. While there are a multitude of pseudo-random number generators (PRNG) readily accessible, not all of them are suited for multiprocessing. A number of different PRNGs were applied to simulations and evaluated in this study. For a discussion regarding random number generators and their compatibility with multiprocessing applications see section 2.2.

The implementation was based on developments within the Unstructured Reaction Diffusion Master Equation (URDME) framework. The URDME framework provides an interface to external geometry and mesh handling in Comsol Multiphysics, furthermore it provides logical separation of the core simulation routines. A Matlab interface is used to connect the simulation routines and the mesh handling software[4].

By analyzing the execution times of both the serial and parallel implementation in a given simulation environment an eventual decrease in execution time for the parallel implementation can be concluded. Furthermore the performance
of a number of different PRNGs were profiled in the context of multiprocessing. The original serial implementation of Gillespie’s direct algorithm was supplied by the URDME source code under the GPL3 license.

1.1 Related work

Drawert, Engblom and Hellander[4] stated that stochastic simulation of reaction kinetics has emerged as an important computational tool in molecular systems biology and will likely continue to grow in importance as experimental techniques are further developed and spatial models can be calibrated to biological data.

Furthermore, in cases for which mean-field analysis has been shown to be insufficient, stochastic models provide a more accurate and computationally manageable alternative[5][6][7].

Östberg et al.[8] argued that computational researchers and developers of scientific software often are aware of parallelization issues but lack the detailed knowledge of issues involved in computational enactment in distributed environments. As such they argue that design patterns and integration tools that help overcome barriers can be expected to have great impact in many areas in eScience. Developments have such been made in URDME server modules which enables remote execution using distributed computing. This allows highly task parallel investigations to use distributed resources such as clusters.

Many applications require a large number of sample realizations to be generated in order to allow for useful statistical analysis. Exploring different parameters or estimating responses to stimuli adds further complexity such that the generation of tens of thousands of independent realizations is not uncommon. For these applications computational efficiency is an important concern[4].

The methods presented in this study takes a step towards improving computational efficiency and adapting simulation algorithms in URDME to be more suitable for distributed computing environments such as presented by Östberg et al.[8].
2 Background

2.1 Unstructured Reaction-Diffusion Master Equation (URDME)

The Unstructured Reaction-Diffusion Master Equation framework, also known as URDME, serves as a layer of abstraction between the core simulation routines and the mesh handling software. The URDME framework consists of three separate layers with interfaces between them. At the top level some third party software for mesh generation is used, presently URDME provides an interface towards Comsol Multiphysics. The middle layer is in Matlab, it allows for model construction and connects the top layer with the bottom layer. At the bottom layer are the solvers, these are usually written in a lower level compiled language for computational efficiency. A solver is a program which carries out a simulation algorithm given any model data and returns the solution trajectory. Presently URDME provides a handful of core solvers which are ready to be used directly and/or modified. It also supports integration of plugin solvers with the proper extensions at the middle layer as well as PDE Toolbox. Because the complexity of handling the geometries are hidden, the core simulation routines can easily be modified or extended. It is worth noting that both the top and bottom layer may be replaced as long as the middle layer is extended accordingly[4].

![Figure 1: Shows the three logical layers of the URDME framework](image-url)
In order to carry out a simulation the geometry must be defined at the top level from a software such as Comsol Multiphysics, this is then exported into Matlab via the interface provided by URDME. The user needs to supply a model file that contains data related to the simulation such as initial distribution of molecules, simulation interval etc. Other files the user can supply are propensity files, these are C programs that defines the propensity functions for the chemical reactions used in the model. These may also be automatically created. Once the previous steps are completed URDME can execute the simulation. This is done by invoking the urdme function in Matlab. From this point on everything is conducted automatically and when the simulation completes the urdme function returns a modified model data structure with a stochastic solution trajectory file attached to it\cite{4}.

2.1.1 Solvers

The solvers carry out the simulation routines and as such are they are most liable for potential optimization. In order to carry out the simulation routines they are passed model data from the middle layer in Matlab. Calling the mex function in Matlab with the C program file as input will compile and create a MEX file. MEX files can be executed in Matlab by calling feval with the MEX file as input and behave similarly to a Matlab function\cite{9}.

When the simulation routines finish computation the trajectory of the system is stored in a data structure that is readily accessible by the middle layer due to the mex interfacing.

Solvers in URDME consists of two main components, a makefile used for compiling and building the solver automatically and the solver source files. The solver in this study is written in ANSI-C and uses GNU style makefiles.

2.1.2 Stochastic Simulation Algorithm (SSA)

In this section the underlying algorithm of the SSA solver that is used to carry out the simulation is presented.

SSA and Gillespie’s direct algorithm are two names commonly used interchangeably to refer to the method developed by Daniel T. Gillespie in 1977\cite{3}. The core solver in URDME refers to it as the SSA solver, hence in the context of the solver it will be referred to as SSA. In all other contexts it will be referred to as Gillespie’s direct algorithm.

The algorithm works by iterating independently over sub-volumes, simulating and determining which reaction happened in what sub-volume at a specific time step and updates the molecule counts accordingly. The solver also supports iterating over a number of independent replicas. A replica is usually a copy of the system that is being simulated but can include a different initial distribution of molecules or is set to use a different seed for the random number generation.

The algorithm begins by initializing all of the necessary variables as well as initializing the RNGs to be used. Once initialization has taken place the algorithm proceeds to the Monte Carlo step.
In the Monte Carlo step the algorithm will first generate a random number. This random number is then used to decide which reaction that occurs in this time step. The decision of which reaction took place is dictated by the supplied propensity functions and the state of the system, thus the probability of a given reaction is proportional to the amount of molecules in the volume. Another random number is generated and used to determine the next time step for the system. The algorithm proceeds to increase the time and update the molecule count based on the reaction that was determined in the previous step.

This process continues to iterate until some stopping criterion is met such as the time reaching the end of the time interval of the simulation.

2.2 Random number generators

The generation of random numbers is one of the most time consuming factors in (Monte Carlo) simulations hence the choice of pseudo random number generator (PRNG) is critical not only in terms of quality but also in terms of resources[10]. There are a multitude of techniques for generating pseudo random numbers and developing such techniques is still an active topic of research. This section provides an overview of random number generators followed by a discussion of a handful of particular methods that were considered in this study together with their advantages and/or disadvantages.

PRNGs relies on a state variable to determine the next number in a sequence. Whenever a random number is sampled from the generator, it proceeds to also update it’s state in a deterministic manner. Given the same initial state the generator will provide the same sequence of numbers every time[10]. Despite the deterministic behaviour of PRNGs they are still useful for simulating randomness. The method for obtaining the next number in the sequence is designed to provide numbers distributed over some interval such that they pass certain statistical tests and does not exhibit any easily discernible pattern[10]. As such the behaviour of a sequence of numbers generated by a PRNG given some arbitrary seed aims to behave similarly to a sequence of truly random numbers.

Below a number of different PRNGs are presented.

2.2.1 drand48 and cache contention

This random number generator is included in the C standard library and stores its state in an internal buffer[11]. Presently the serial SSA solver uses drand48 as its source of random numbers.

Storing the state in an internal buffer will prove problematic in the context of multiprocessing. Having multiple threads on different processor access the state in the internal buffer will lead to a performance degrading pattern known as cache contention. More precisely, cache contention occurs when two or more CPUs alternately and repeatedly update the same cache line[12]. In this case it occurs due to true sharing where another thread has written to the same
memory address that the current thread is referencing. This leads to a cache miss which incurs the aforementioned performance degradation.

Other alternatives that do not suffer from performance degradation when used in a multiprocessing application are described in the coming sections.

2.2.2 rand_r

This is another random number generator included in the C standard library, rand_r. It differs from drand48 in that it does not rely on an internal buffer for its state. Instead it takes an address to a seed data segment as an argument and updates it when sampling from the generator[13]. This allows for different threads to maintain private states thus avoiding both corrupting the data and cache contention. This leads to the conclusion that rand_r will not suffer from the shortcomings of drand48 that was discussed in the previous section. This is beneficial for the multi processing application since large performance gains are possible by avoiding cache contention.

2.2.3 GNU Scientific Library (GSL)

The GNU Scientific Library provides several algorithms for generating random numbers that allow each instance to maintain it’s own state making them safe for use in multiprocessing[14]. The GSL random number generators will thus, like rand_r avoid cache contention. For a comprehensive list of the random number generation methods supported in GSL see the GSL documentation[14]. In this study, a few select algorithms were tested through the GSL library, namely:

- gsl_taus2
- gsl_mt19937
- gsl_ranlxs0
- gsl_ranlxs2

2.3 Multiprocessing

Multiprocessing involves the use of multiple CPUs within a single computer system and can be exploited to various degrees depending on the structure of the program. In the most trivial example the program is so-called embarrassingly parallel and multiple independent threads can be spawned. These types of programs will generally gain significant performance increases from parallelization.

There are potential hazards involved when executing a program in parallel. The order of execution is no longer guaranteed which can lead to race conditions and having multiple processes access and modify data in the same address space leads to data races.

The most common method historically for solving race conditions and data races is using locks to block threads from executing until some data is available. This introduces performance overheads so it should be avoided where possible.
The serial implementation of the SSA solver would however suffer from data races if it was simply converted to run simulations in parallel without further modifications. This happens due to the multiple sub-volumes sharing the same address space for reading and writing their intermediate states. A simple solution to this problem is presented in section 3.

The SSA solver being analyzed in this study happens to fall under the category of being embarrassingly-parallel over the independent replicas. Parallelism can thus be exploited to a high degree over replicas in a straightforward manner. Parallelizing over the subvolumes will not be as straightforward as race conditions related to random number generation has to be avoided. Using a single RNG producing a sequence of numbers would not give consistent results because of the sequence being dependent on the order of execution. A solution to this problem is presented in section 3.2.2. and further detailed in 4.1.2.

2.3.1 Open Multi Processing (OpenMP)

OpenMP is an API that provides shared-memory multiprocessing programming in C, C++ and Fortran. OpenMP uses a fork and join model to implement multi threading as a method of parallelization. The master thread forks a number of threads and includes them in a team. The workload will then be divided to members of the team. OpenMP acts at compile time and thus interfaces via compiler directives. The run time environment then allocates threads to processors according to the state of the environment at that point in time. [15].

2.3.2 Expected performance

One might expect an N-times speedup on a platform with N processors. This turns out to be a naive assumption since there are a number of concerns which negatively affects the speedup when running a parallelized program. For starters there is a theoretical limit on the expected speedup that is governed by Amdahl’s law, which states that the theoretical upper limit of speedup is limited to the fraction of the program that can be run in parallel[16].

Furthermore there may be data dependencies between the threads, in which case the threads would need to communicate and potentially lock themselves until the required data is available.

Other factors which impact performance include the overhead for thread creation and load balancing. Compiler optimizations might be lost due to decreasing transparency when using compiler directives such as with OpenMP.
3 Proposed Solution

The proposed solution is to modify the current serial implementation of the SSA solver present in the URDME framework to exploit parallelism. As has been described the algorithm presents desirable properties for parallelism. In this solution parallelization over the simulation loop is proposed by using OpenMPs compiler directives for distributing workload across threads.

As previously described the choice of random number generator will be of importance both in terms of performance and quality of the results. As such a random number generator interface is proposed. This interface provides simple functions to initiate, sample and destroy a random number generator of choice. The goal of the RNG interface is to provide easy testing of the different RNGs to eventually decide a default for the parallel solvers.

3.1 Random number generation

For easy selection of different random number generators a RNG interface is proposed. This interface provides a handful of functions to initiate, seed, sample and destroy a selected RNG. The interface allows for selection of RNG by setting an environment variable which is done automatically at the Matlab layer. As such it is proposed that selection of the RNG is implemented as an input argument to the `urdme` function like such:

```matlab
model.rng = 'DRAND48';
model.seed = 123;
urdme(model);
```

Where the `rng` property acts as a compile time argument to the solver. This also means that recompilation is required whenever a new RNG is selected. The above example would compile and run the solver with the `drand48` algorithm using the seed 123. Regarding the options of RNGs, the proposed ones are those detailed in section 2.2.

The functions for seeding, sampling and destroying the RNG are defined using compiler directives such that they are substituted using inline expansion at compile time for optimization purposes. The function for initializing the RNG is a runtime function due to technical limitations of inline expansions. A RNG object is created when first initializing the RNG and is then passed on for sampling and destroying the RNG.

3.2 Multiprocessing

The SSA solver uses a state and a rate vector to keep track of the state of a subvolume. In the serial implementation it sufficed to allocate one state and rate vector and reuse it in each subvolume. This proved to be problematic when running in parallel since it incurs a data race. In order to avoid this data race the proposed solution is to allocate thread-private state and rate vectors for
each thread. This will introduce a larger memory overhead but will avoid the need for locking threads which would be fatal for performance.

In this section two different solutions are proposed, the first solution follows URDME’s seeding syntax and relies on executing the subvolumes within a replica in order. The other solution proposes a new seeding syntax which allows for out of order execution of the subvolumes thus allowing parallelization over both the replicas and subvolumes.

3.2.1 Task based solver

This solution follows URDME’s present seed syntax and is described below. Here a number of threads are spawned and divided over the replicas. Each thread over the replicas proceeds to initiate an RNG and generate tasks for the computation of subvolumes. The tasks are put into a queue and then fetched and executed when a thread is available. Queuing and fetching threads are handled by utilizing OpenMP. Due to the desired reproducibility of the simulations the tasks have to be fetched in order from the task queue to ensure that the state of the RNG is not corrupted by out of order sampling.

3.2.2 Transparent solver

In this solution it is proposed that parallelization occurs over both the replicate cases and subvolumes. Parallelizing over both domains involves a different method for seeding the RNGs which leads to the new seeding syntax further described below. This solution relies on collapsing the simulation loops over the replicate cases and subvolumes into one larger loop. Whereas previously seeding took place once for every replicate case, seeding will now have to take place once for every work unit. To get an even spread of seeds, another RNG is used is used to generate seeds for the primary RNG. The secondary RNG takes garbage data generated from the original seed and subvolume as input. Using this method for generating seeds lowers the probability that two work units are seeded equally while remaining reproducible.

This proposal favourably involves modifying the serial solution to map the random numbers generated to those generated using the method presented here.
4 Implementation

For the parallel implementations of the SSA solver OpenMP was used. A random number generation interface was implemented to easily compile the solver with different RNG methods. Support for a number of different RNGs was implemented, namely: \texttt{drand48}, \texttt{rand} and a handful of GSL random number generators. The parallel implementations of the SSA solver is presented followed by the RNG interface. Lastly the extensions made to the middle layer in Matlab is presented.

4.1 SSA solver

The parallel SSA solvers were based on the serial solution included in the URDME source code licensed under GPL3. Parallelization was achieved by using the OpenMP library for spawning threads and distributing workload amongst them.

For both solutions configuration of the number of threads to be used in the simulation is supplied as an input argument to the solver function and can be passed to the solver through Matlab by modifying the \texttt{solverargs} property.

To avoid data races the state and rate vectors were implemented to be thread private where as previously only one state and rate vector was respectively allocated and reused.

Reporting is handled within a critical section due to limitations of Matlab and OpenMP which leads to undefined behaviour when executing \texttt{mex} functions in a parallel section.

Reaction statistics are collected by atomically incrementing a reaction counter for every iteration. When the computation is finished the trajectory is attached to the return value and passed to the Matlab layer.

4.1.1 Task based solver

This solution follows URDMEs RNG seeding syntax and was implemented using OpenMPs task directives for creating a task for each subvolume. The tasks are then put into a queue and fetched whenever a thread becomes available.

The algorithm exploits the independence of replicas and spawns a number of threads that each produce tasks simultaneously. Should there be more threads available than there are subvolumes the task directive is ignored and the algorithm will execute the task immediately instead. This is done to avoid unnecessary overhead from load balancing threads. For every replica a random number generator is created and seeded according to the seed specified at the Matlab layer.

As previously mentioned the threads are put into a queue and are then fetched by available threads. To avoid corrupting the RNGs state the tasks are specified to contain a dependency in the state of the RNG. This avoids corruption of the RNG state that would happen if an out of order sample would occur. This is illustrated in the pseudo-code on the next page.
```c
#pragma omp parallel
{
    #pragma omp for
    for(int k = 0; k < Nreplicas; k++){
        /* Create and seed RNG */
        rand_state_t *rng = init_rng();
        seed_rng(rng,k);
        for(int subvol = 0; subvol < Ncells; subvol++){
            #pragma omp task if(threads < Nreplicas) depend(out : rng)
            {
                /*Simulate */
            }
        }
    }
}
```

The task directive is then handled by OpenMP’s scheduler to ensure in-order execution within a replica. Subvolumes has to be executed in order and thus no performance gain is expected for a single replica, instead there is possibility for a performance decrease as thread locking and load balancing overheads might become significant. As such this solution might not utilize all available threads at all times.

### 4.1.2 Transparent solver

This solution uses a new syntax for seeding the RNGs and uses a loop collapsing technique for parallelizing over both the replicate cases and the subvolumes simultaneously.

By introducing a new seeding syntax reproducibility can be retained while allowing out of order execution of the subvolumes.

To avoid data races multiple RNGs are created and used throughout the simulation. The amount of RNGs created will correspond exactly to the amount of threads specified and thus each thread will use a private RNG. The amount of threads is specified in the compile time definition NTHREADS. The solution is illustrated by the pseudo-code below.

First a number of threads corresponding to the specified amount of threads are created and stored in the data segment. The RNGs are then initialized only upon the first call to this function. This is done to reduce overhead when use cases involve switching back and forth between the solver layer and Matlab layer frequently, such as would be the case for certain spatial coupling implementations.

```c
static rand_state_t *rngs[NTHREADS];
if(*rngs == NULL){
    for(int i = 0; i < NTHREADS; i++)
        rngs[i] = init_rng();
}
```
The algorithm then moves on to the simulation step, this is illustrated below:

```c
#pragma omp parallel for
for(size_t ij = 0; ij < Ncells*Nreplicas; ij++){
    size_t subvol = ij % Nreplicas;
    int k = (int) ij / Nreplicas;

    /* Select RNG*/
    rand_state_t *rng = rngs[omp_get_thread(num)];
    /* calculate hash value based on seed and subvol number */
    unsigned int hash = (seed_long[k]+subvol)*2654435761 % 2^32;
    /* seed rng with a random value generated from the calculated hash */
    seed_rng(rng,rand_r(&hash));

    /* Simulate */
}
```

The loops which was previously nested has here been combined into one big loop which iterates over work units. Each work unit corresponds to the simulation of a subvolume inside a replica and is seeded individually. To ensure that the seeds are reasonably spread the `rand_r` algorithm generates a seed for the primary RNG using a hash value generated by the supplied seed and subvolume number. The hashing function utilizes Knuth’s multiplicative method [17] which uses the golden ratio of $2^{32}$ to ensure that they do not share any factors. This aims to provide further security and minimizing the chance of using the same seed in multiple subvolumes while remaining reproducible. This loop is transparent to the scheduler hence why it’s called the transparent solution.

The implications of this solution is that subvolumes no longer need to be simulated in-order for the algorithm to provide reproducibility. This leads to parallelism being utilized to a larger degree and allows for further exploitation of the independence between subvolumes presented by SSA.

### 4.2 RNG interface

The RNG interface is implemented in C and provide generalized functions for initiating, sampling and destroying a random number generator of choice. The choice is specified by defining `URDMERNG` at compile time. In gcc this can be done by compiling with the flag:

```
-DURDMERNG=x
```

where `x` is the RNG code. A table of defined constants (RNG code) and their respective methods are presented below.

To compile the solver with the `rand_r` method one would either include the compiler flag:

```
-DURDMERNG=RAND_R
```
Table 1: Shows the defined constants, also known as RNG codes and their respective method. For a better description of each RNG see section 2.3.

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAND48</td>
<td>1</td>
</tr>
<tr>
<td>RAND_R</td>
<td>2</td>
</tr>
<tr>
<td>GSL_TAU2</td>
<td>3</td>
</tr>
<tr>
<td>GSL_MT19937</td>
<td>4</td>
</tr>
<tr>
<td>GSL_RANLXS0</td>
<td>5</td>
</tr>
<tr>
<td>GSL_RANLXS2</td>
<td>6</td>
</tr>
</tbody>
</table>

It is worth noting that this is not required to be done manually by the user. Instead the RNG selection is controlled at the middle layer as described in section 4.3.

In order to reduce overhead related to function calls the functions for seeding, sampling and destroying RNGs is defined at compile time depending on the defined RNG code. The performance benefit of this is expected to be small on modern hardware. However, seeding and sampling from the random number generator will be frequent and thus this optimization was included.

The function for initializing the RNG was implemented as a runtime functions that selects a method for initialization depending on the RNG code specified at compile time. This was done due to technical limitations related to inline expansions. Initializing RNGs is expected to be an infrequent event and as such will not likely introduce significant performance overheads.

Using the RNG interface involves the following steps; first the RNG is initialized by calling `init_rng`. `init_rng` returns a pointer to an object of type `rand_state_t` which can then be passed on to the `sample_rng`, `seed_rng` and `destroy_rng` functions. Setting the seed of the RNG is done by invoking `seed_rng` with the desired seed. Calling `sample_rng` returns a value between 0 (inclusive) and 1 (exclusive). Once no further sampling is needed `destroy_rng` can be called for deallocation.

When compiling the program the appropriate compile time definitions needs to be supplied as described above. If no definition is present at compile time the RNG method will default to `drand48`.

### 4.3 URDME extensions

In order to accommodate the extended behaviour at the Matlab layer some extensions were made. The function which carries out the simulation routine is invoked in Matlab by calling the `urdme` function. This function takes a number of inputs from the model file and proceeds to compile and run the solver. Configuration of the number of threads is done in the model file and was
accommodated by sending it to the `urdme` function as an argument to the makefile. This is specified in the `makeargs` property. Specification of the number of threads are specified in the model file as:

```python
model.makeargs = {'nthreads', 4}
```

To accommodate the selection of random number generator further extensions was made at the Matlab layer. The `urdme` function was extended to accommodate a RNG property. By setting the RNG property the program will be compiled with the desired RNG algorithm. The value of the property is expected to follow the RNG code notation described in table 1. Setting the RNG property can be done in the model file as follows:

```python
model.rng = 'RAND_R'
```

Currently the RNG syntax is not supported by any other solvers than SSA in URDME and will throw an error if attempting to use an RNG that is not DRAND48 (the default supported RNG).
5 Evaluation

In this section the simulation environment is detailed before the performance results using the different RNG algorithms shown in section 2.2. is presented. This is followed by an overview of the difference in execution times between the serial and the parallel solvers.

5.1 Simulation

To evaluate the performance the new parallel solver and the serial solver was applied to a simulation. The simulation of choice for evaluating performance was simulating morphogenesis in 2 dimensions which has been studied in URDME previously[18] and included as an example in the source code.

Morphogenesis is the process which causes an organism to develop it’s shape. It does so by controlling the spatial distribution of cells during the development of an organism. It has been argued that reaction-diffusion processes are the underlying mechanism for morphogenesis[19]. The model to be simulated was provided in the URDME repository and uses a torus mesh and can be seen in Figure 2.

In order to evaluate the performance of the different RNGs both the parallel and the serial solver were applied to a simulation using each of the RNG algorithms presented in section 2.2.

For the parallel simulations the execution times were profiled using different number of threads. The number of threads used in the simulations were varied from 1 to 12 in order to observe the effect of parallelization with an increasing number of threads. For each simulation OpenMP was set to be dynamic.

Profiling was done at the Matlab layer using the built-in Matlab function \texttt{profile} which records execution times of the program following the \texttt{profile} statement and outputs the result in a HTML file. Because the profiling results of the total execution time include execution times for pre- and post processing steps a simple bash script was used to single out the execution time of the solver algorithm.

The simulations were carried out within Matlab R2020a where as compilation was performed using gcc 6.5.0 using level 3 compiler optimizations. For parallelization OpenMP version 4.5 was used.

All of the simulations were performed on an Intel Core i7 8700K 3.7 GHz processor with 12 available threads running the Ubuntu 18.04 LTS operating system. The results are presented in the next section.
5.2 Multiprocessing RNG performance

Here the results from simulating the example described in section 5 with the different RNG methods described in section 4.2. are presented. The results were obtained by running the simulation using the task based parallel solver with 4 threads and 4 replicas.

![Execution times of RNG algorithms](image)

Figure 3: Shows the execution times in seconds of the respective RNG algorithms.

No significant performance difference is observed between RAND,R, GSL,TAUS2 and GSL,MT19937. The execution times for DRAND48 is significantly higher than the aforementioned. This is likely attributed to the occurrence of cache contention discussed in section 2.2.3. Furthermore, the execution time of GSL,RANLXS0 and GSL,RANLXS2 is observed to be higher than those of RAND,R,GSL,TAUS2 and GSL,MT19937. This is likely attributed to differences in the methods for obtaining random numbers.
Both the task based and transparent solution appears to follow the same pattern using the different RNGS. It appears that the execution times are marginally higher for the transparent implementation versus the task based

5.3 Serial vs. parallel

In this section the resulting execution times of running the simulation using the two parallel solvers on a variable amount of threads and replicas is presented. This is then compared to the execution times of the serial solver.

In the following figures, the execution time of the serial solver is displayed as a horizontal line to provide easy comparison.

![Execution times of the different solvers running on a varying amount of threads](image)

Figure 4: Shows the execution times in seconds of a simulation running on 1 replica. All solvers used the GSL\_MT19937 RNG.

As can be seen in Figure 4 the execution times of the task based solver is similar to that of the serial solver for the 1 replica case. This is expected due to the lack of concurrent execution within a replica. For the transparent solver there is a noticeable performance increase which is in line with the narrative presented in section 4.1.2.

Because the transparent solution is able to exploit parallelism over both replicas and subvolumes a noticeable performance increase is observed even for a single replica. The speedup is drastic between one thread and four threads where after the performance gains seems to pan out with further increasing threads.

The speedup of the transparent solution running on 12 threads and 1 replica was observed to be around 3.14x faster than the serial solver. The task based solver was observed to be marginally slower than the serial for this simulation.
Figure 5: Shows the execution times in seconds of a simulation running on 6 replicas. All solvers used the GSL\_MT19937 RNG.

Figure 6: Shows the execution times in seconds of a simulation running on 12 replicas. All solvers used the GSL\_MT19937 RNG.
As the amount of replicas increase the performance benefits of the task based solver starts becoming more apparent as can be seen in Figure 5 and 6 where the simulation uses 6 and 12 replicas respectively.

For the 6 replica simulation results presented in Figure 5 both the task based and parallel solver seem to follow the same pattern in terms of speedup. The speedup of the transparent- and task based solution running on 12 threads and 6 replicas were observed to be around 3.23x and 2.96x respectively.

Similarly for the 12 replica simulation results presented in Figure 6 both the task based solver and the transparent solver seemed to follow the same pattern in terms of speedup. The speedup of the transparent- and task based solution running on 12 threads and 12 replicas were observed to be around 3.32x and 3.63x respectively.
6 Discussion

Profiling results indicate that both the parallel solvers present significant performance improvements for the multiple replica case. As can be seen in Figure 5 and 6 both the task based and transparent solver seem to follow the same pattern in terms of speedup where the performance benefits are gained quickly up to 4 threads where after the increase per thread appears to stagnate. The transparent solver was shown to be faster in the 6 replica/12 threads case with a speedup of 3.23x compared to the 2.96x of the task based solver. This difference is likely attributed to the ability of the task based solver to parallelize over the subvolumes. For the 12 replica/12 threads case the transparent solver was shown to be slower with a speedup of 3.32x compared to the 3.63x of the task based solver. The difference is likely attributed to higher performance overheads due to load balancing in the transparent solver.

In the single replica case the task based solver performs similarly to the serial solver. This was expected due to the solvers inability to parallelize over subvolumes. The transparent solver on the other hand presents a significant performance increase with speedups reaching 3.14x on 12 threads on the local machine.

Observations in Figure 5 shows that there is a small difference in execution times for the task based and transparent solution when running a simulation using 6 replicas and 12 threads. In this particular simulation the worst case scenario for the task based solver would result in 6 of the threads being under utilized due to data dependencies. The results points towards a small performance advantage of about 9.1% for the transparent solution. The observed difference do not correspond to predictions of the worst case scenario where 50% of the threads would be under utilized for the task based solver. This leads to the conclusion that the task based solver manages to maintain utilization across the available threads to a reasonable degree throughout the simulation. A possible explanation is that OpenMP is cleverly balancing the load across the threads. Seeing as every thread has a choice of generating tasks or executing them, it is reasonable to believe that OpenMP balances the amount of threads that are generating and executing respectively to maximize utilization.

It was observed that a bottleneck was reached already when executing on 4 threads for both solvers when simulating multiple replicas. Possible explanations for this are memory bandwidth and I/O bandwidth limitations.

Regarding the RNGs, four of them presented promising execution times in the multiprocessing context, namely; rand_r,gsl_taus2,gsl_mt19937 and gsl_ranlxs0. Early on rand_r was scrapped as an alternative due to its limited spectral properties[13]. All of the tested GSL RNGs are recommended for usage in simulation by the GSL team and the second generation of RANLXS, referred to as gsl_ranlxs2 here, have the strongest proof of randomness[14]. While the gsl_ranlxs2 algorithm might provide the highest quality of random numbers, the execution times were significantly higher for this algorithm as presented in Figure 3. As observed in Figure 3 the gsl_mt19937 RNG provided fast execution times in the multiprocessing context. Furthermore the gsl_mt19937 RNG has
passed the DIEHARD statistical test and provides a large period[14].

Regarding the state vectors a comparison can be seen in the table below:

<table>
<thead>
<tr>
<th>RNG</th>
<th>Size of state vector [bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAND48</td>
<td>48</td>
</tr>
<tr>
<td>RAND_R</td>
<td>16</td>
</tr>
<tr>
<td>GSL_TAUS2</td>
<td>48</td>
</tr>
<tr>
<td>GSL_MT19937</td>
<td>9984</td>
</tr>
<tr>
<td>GSL_RANLXS0</td>
<td>384</td>
</tr>
<tr>
<td>GSL_RANLXS2</td>
<td>384</td>
</tr>
</tbody>
</table>

Table 2: Shows the size of the state vector in bits for each of the RNG algorithms.

The *gsl_mt19937* generator has a much larger state vector size which could be problematic for the task based solver. It is expected to be manageable for the transparent solver, since it won’t create nearly as many RNGs as the task based solver would.

The performance difference between the transparent solver and task based solver was small for the multiple replica case. However, unlike the task based solver, the transparent solver showed a performance increase in the single replica case which was expected due to the solvers ability to parallelize over the subvolumes. It should be noted that the transparent solver introduced a new syntax for seeding the RNGs and as such it might lead to inconsistencies if it were to coexist with the serial solver with a separate syntax. Optimally, the serial solvers random numbers would be mapped to those generated by the transparent solver were they to coexist. Due to the structure of the transparent algorithm the serial solver could be replaced with the transparent solver without loss of functionality even if OpenMP is omitted at compile time.

Regarding the task based solver, while it followed the URDME syntax present in the serial solver it was deemed obsolete in comparison to the more versatile transparent solver. Thus it was deemed that the transparent solver would be most suitable for integration into the URDME code base, either as a separate, plug-in solver using a different syntax or as a replacement of the serial solver.

Thus, due to the ability to generate quality random numbers required by scientific applications as well as the fast execution times, the *gsl_mt19937* algorithm was selected to be default for the parallel solvers despite it’s large state vector size. It is expected to be manageable since the amount of RNGs allocated will only correspond to the amount of threads that are being used throughout the simulation.
7 Conclusions and future work

Two new solvers were proposed and implemented to exploit parallelism within the SSA algorithm present in the URDME framework. The first solver used a technique of generating tasks of subvolumes and putting them in a queue where they then are fetched and executed in order within a replica. This solver was named the task based solver. The other solver used a technique for collapsing the simulation loops over the replicas and subvolumes into one larger loop. This solver was named the transparent solver and introduced a new syntax for seeding the RNGs to ensure reproducible results even when subvolumes are computed in an arbitrary order.

For a single replica the transparent solver presented performance speedups up to 3.14x on the 12 thread local machine whereas the transparent solver performed similarly to the serial solver regardless of the number of threads. This was expected due to the task based solver requiring that subvolumes are computed in order.

In the multiple replica case (6 and 12 replicas) both solvers presented a similar pattern in terms of speedup. Running on 12 threads, the transparent solver presented speedups of 3.23x and 3.32x for 6 and 12 replicas respectively whereas the task based solver presented speedups of 2.96x and 3.63x for 6 and 12 replicas.

A number of RNG methods were tested in a multiprocessing context. The previously used `drand48` presented slow execution times likely due to cache contention. Other RNG methods tested were `rand`, `gsl_ranu2`, `gsl_rngmt19937`, `gsl_ranx0`, `gsl_ranx2`. Ultimately the `gsl_rngmt19937` RNG was selected to be the default for the parallel solvers due to its fast execution times and high quality random numbers.

Finally, the transparent solver was deemed most appropriate for integration into the URDME code base due to its more versatile method of parallelization which presented performance increase for both the single- and multi replica cases. It was argued that the serial solver could be replaced with the transparent solver without loss of functionality even when OpenMP is omitted at compile time. Thus the transparent solver could either be integrated as a separate plugin solver or serve as a replacement of the serial solver.

For future work the presented method could be paired with a spatial coupling algorithm. This would involve executing the spatial coupling algorithm on the product of the SSA algorithm in each time step. Because the spatial coupling algorithm would be significantly more difficult to execute in parallel the performance improvements in the non spatially-coupled algorithm presented here would provide performance benefits in such applications.

Further performance improvements are to be had were it possible to resolve the current bottleneck. The bottleneck could be related to limitations of memory bandwidth and I/O bandwidth and thus investigating this further could be of interest in the future.

The presented methods could be also be of interest for distributed computing which has previously been studied within the context of URDME[8].
8 References

References


