Vector Operations for Accelerating Expensive Bayesian Computations – A Tutorial Guide[*]

David J. Warne[‡,§,†], Scott A. Sisson[¶], and Christopher Drovandi[‡,§]

Abstract. Many applications in Bayesian statistics are extremely computationally intensive. However, they are often inherently parallel, making them prime targets for modern massively parallel processors. Multi-core and distributed computing is widely applied in the Bayesian community, however, very little attention has been given to fine-grain parallelisation using single instruction multiple data (SIMD) operations that are available on most modern CPUs. In this work, we practically demonstrate, using standard programming libraries, the utility of the SIMD approach for several topical Bayesian applications. Using the C programming language, we show that SIMD can improve the single-core floating point arithmetic performance by up to a factor of $6\times$ compared scalar C code and more than $25\times$ compared with optimised R code. Such improvements are multiplicative to any gains achieved through multi-core processing. We illustrate the potential of SIMD for accelerating Bayesian computations and provide the reader with techniques for exploiting modern massively parallel processing environments.

MSC2020 subject classifications: Primary 65C60; secondary 97K80.

Keywords: vectorisation, single instruction multiple data, advanced vector extensions, approximate Bayesian computation, weakly informative priors, sequential Monte Carlo.

1 Introduction

Practical applications in Bayesian statistics are computationally challenging since the posterior density is only known up to a normalising constant. Therefore, advanced Monte Carlo schemes are often required. Intractable likelihoods further compound these computational burdens (Sisson et al., 2018). In addition to posterior sampling, there are other computational challenges in Bayesian statistics. For example, it may be necessary to ensure priors are only weakly informative, and selected from a family of priors with a hyperparameter (Evans and Jang, 2011; Gelman, 2006).

[†]To whom correspondence should be addressed.

[‡]School of Mathematical Sciences, Queensland University of Technology, Brisbane, Australia, david.warne@qut.edu.au

[§]Centre for Data Science, Queensland University of Technology, Brisbane, Australia

[¶]School of Mathematics and Statistics, University of New South Wales, Sydney, Australia

The performance of computational methods and computer hardware continues to improve (Green et al., 2015) largely due to increased parallel processing capacity. To fully exploit modern computational resources for Monte Carlo sampling, implementation details are essential. Code optimisation for hardware acceleration is standard in the *high performance computing* (HPC) discipline, and Bayesian practitioners can benefit from these techniques (Gillespie and Lovelace, 2017).

Many Monte Carlo schemes are inherently parallel. For example, likelihood-free methods, such as approximate Bayesian computation (ABC) require a large number of independent prior predictive samples that can be executed in parallel. However, more sophisticated samplers, such as Markov chain Monte Carlo (MCMC) (e.g. Green et al., 2015; Marjoram et al., 2003), sequential Monte Carlo (SMC) (e.g. Del Moral et al., 2006; Sisson et al., 2007) and multilevel Monte Carlo (MLMC) (e.g. Jasra et al., 2019; Warne et al., 2018) require more effort to efficiently parallelise (Murray et al., 2016) due to synchronisation and communication requirements. These overheads can inhibit scalability for multithreading on central processing units (CPUs).

General purpose graphics processing units (GPGPUs) are highly effective at accelerating advanced Monte Carlo schemes (Klingbeil et al., 2011; Lee et al., 2010a). GPGPUs make heavy use of *single instruction multiple data* (SIMD) (van der Pas et al., 2017), that is, instructions that operate on vectors element-wise. SIMD is also widely available in modern CPUs containing *vector processing units* (VPUs). Such SIMD CPU implementations can achieve performance boosts comparable with GPGPU implementations (Lee et al., 2010b; Mudalige et al., 2016). Therefore, practical understanding of SIMD for CPUs is relevant to applied Bayesian analysis. We focus on code structures and algorithmic techniques for practitioners to harness VPUs available in most commodity CPUs, and provide example codes using both R and C.[1] Since almost all modern desktops and laptops have SIMD capabilities, these techniques are highly relevant to many applied statisticians and not exclusively to users of HPC systems.

The paradigms of parallelism are *distributed computing*, *multithreading*, *vectorisation*, and *pipelining* (Trobec et al., 2018; van der Pas et al., 2017). Each paradigm has a granularity that refers to the ratio of communication to computation for a parallel workload. We refer to distributed computing and multithreading as coarse-grained, whereas vectorisation and pipelining are fine-grained. Most exemplars of fine-grain parallelism in statistics are based on accelerators, such as, GPGPUs (Lee et al., 2010a; Terenin et al., 2018), Intel Xeon Phis (Hurn et al., 2016; Mahani and Sharabiani, 2015), and custom co-processors (Zierke and Bakos, 2010). Since accelerators are not practical for many practitioners, we do not discuss them here. Instead we highlight parallelisation strategies for optimal utilisation of CPUs with a focus on vectorisation using SIMD.

We demonstrate the utility of CPU-based SIMD for accelerating Bayesian inference. In Section 2 we introduce principles of code optimisation and hardware acceleration using SIMD, then practically demonstrate these principles, in Section 3, through a tutorial comparing optimised R and C implementations of prior predictive sampling for ABC (Sisson et al., 2018). Finally, several topical case studies are presented in Section 4: the computation of Bayesian p-values for prior weak informativity tests (Evans and Jang,

---

[1]Example code is available from GitHub: https://github.com/davidwarne/Bayesian_SIMD_examples.

2011), and parameter inference in econometrics (Bekaert et al., 2015). We highlight algorithmic features that are suited for parallelism and demonstrate modifications for practical guidance. The clear demonstration of performance benefits using SIMD for CPUs is our main contribution. Example optimised implementations are provided as Supplementary Material (Warne et al., 2021) using the C language, OpenMP standard (version ≥ 4.5), the Intel Math Kernel Library[2] (MKL) (version ≥ 2018), and the Intel C compiler[3] `icc` (version $\geq 17.0.1$). Our guidelines are also directly applicable to the Julia language (Bezanson et al., 2017) and to external interfaces to pre-compiled C code, such as Matlab C-MEX and Rcpp combined with RcppXsimd (see Section 5).

2   An introduction to code optimisation

Here, we introduce CPU computing concepts essential for optimisation. We avoid technological details in favour of simple tools and rules-of-thumb for practitioners.

2.1   Vectorisation with SIMD

CPU cores are sequential,[4] performing a single operation, such as arithmetic or reading/writing data, per clock cycle. Without optimisations, the loop in Figure 1(a) requires four floating point addition operations. Modern CPU cores can perform scalar or vector arithmetic, thus Figure 1(a) takes only a single *vector* addition ($4\times$ speedup).
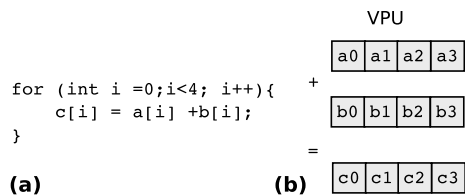

Figure 1: VPUs execute in SIMD. The body of the `for` loop in (a) requires four scalar additions but only one vector addition as shown in (b).

Vector arithmetic is implemented using VPUs that execute in SIMD. The VPU inputs are wider than the scalar arithmetic units. A VPU that accepts 256 bit inputs can store eight single precision (32 bit) or four double precision (64 bit) floating point numbers. VPUs perform arithmetic element-wise on input vectors as shown in Figure 1(b). Acceleration through SIMD is multiplicative with gains achieved through multithreading. Using 18 threads and 512 bit vectors of an Intel Xeon Gold 6140 CPU,[5] could theoretically perform up to $18 \times (512/64) = 144$ times as many double precision operations per second than sequential processing with scalar operations (Tian et al., 2013).

---

[2]https://software.intel.com/content/www/us/en/develop/tools/performance-libraries.html.

[3]https://software.intel.com/content/www/us/en/develop/tools/compilers/c-compilers.html.

[4]Although, they do not necessarily execute operations in the order defined by the users code. CPUs re-order operations as needed to optimise pipelining. This is called out-of-order execution and is a feature that is not readily available on accelerator architectures.

[5]https://ark.intel.com/products/120485/Intel-Xeon-Gold-6140-Processor-24-75M-Cache-2-30-GHz-

2.2 Vectorisation and multithreading with OpenMP

VPUs may be accessed in a variety of ways. One approach is OpenMP, an open standard for automated parallel computing. OpenMP supports parallel computing through *directives* within standard C/C++ (van der Pas et al., 2017) by providing additional information to the *compiler*. For C/C++ programs, a compiler is used to translate the C/C++ source code into binary CPU instructions. OpenMP directives influence this compilation process. For example, the directive `#pragma omp simd` (Figure 2(a)) guides the compiler to include instructions that utilise VPUs.

```
                            #pragma omp parallel
                            {
#pragma omp simd                #pragma omp for          #pragma omp parallel for
for (int j=0;j<N;j++){          for (int j=0;j<N;j++){   for (int j=0;j<N;j++){
    c[j] += a[j]*b[j];              c[j] += a[j]*b[j];       c[j] += a[j]*b[j];
}                               }                        }
                            }
    (a)                         (b)                          (c)
```

Figure 2: Example OpenMP directives: (a) loop vectorisation, (b) loop parallelisation with multithreading, and (c) shorthand syntax for multithreading.

Using `#pragma omp parallel` tells the compiler to create a pool of threads within the next code block. Within a parallel block, `#pragma omp for` distributes loop iterations across these threads. Figure 2(b) and (c) show the same loop as in Figure 2(a), but parallelised across cores. Combining the use of `simd`, `parallel` and `for`, and `parallel for` is key to performance. The following guidelines are useful:

1. Use `simd` (Figure 2(a)) when loop iterations are independent without conditionals. The loop should only use arithmetic and standard functions (e.g. `sin`, `exp` or `pow`).

2. Use `parallel for` (Figure 2(c)) when loop iterations are independent but utilise conditionals. The loop can be complex and include user function calls.

3. A `simd` (Figure 2(a)) loop can be included within the body of a `parallel for` (Figure 2(c)) loop, but the converse is not true.

4. The full `parallel` and `for` (Figure 2(b)) construct can be useful to more explicitly control the behaviour of individual threads through the use of OpenMP functions.

These guidelines are not strict rules, for example, user defined functions may be called within a `simd` loop provided the function has been appropriately constructed and defined with `declare simd`. Many other features are also available within OpenMP, and we refer the reader to the specifications[6] for details. Trobec et al. (2018) and van der Pas et al. (2017) also provide details on parallel computing and HPC. Alternatives to OpenMP are discussed in Section 5.

---

[6]See https://www.openmp.org/specifications/ for full OpenMP specifications.

2.3   Memory access and alignment

To fully exploit SIMD, it is crucial to manage memory access. Commodity random access memory (RAM) bandwidth is around 20 GB/sec, whereas the VPU can process floating point data at a rate of 600 GB/sec. That means only 3% utilisation is possible for data retrieved from RAM. CPUs avoid this bottleneck with a hierarchy of memory caches, typically with three levels: L1, L2, and L3. Lower cache numbers correspond to higher bandwidth, but smaller capacity. L1 cache can keep the VPU close to 100% utilised, but is typically less than 30 KB. When data are requested, the caches are tested in order. If the data are in L1 cache, then there is no memory access penalty. If not in L1 cache, then L2 is checked and so on. RAM is accessed only when the data are not in any cache, which is called a cache miss. For efficiency, data arrays should be accessed in patterns that minimise cache misses. Aim to reuse data soon after it was last accessed and avoid random memory access patterns in favour of regular access. That is access `a[i]`, then `a[i+n]`, then `a[i+2*n]`.[7] Ideally, `n = 1` or is small to avoid cache churn.

Memory alignment is also important for SIMD. Memory is partitioned into blocks called cache lines. For many recent Intel CPUs, these are 64 bytes in length.[8] When the CPU loads data at an address, the entire cache line is loaded. When doing SIMD, the first byte of a data array should be aligned with the 64 byte cache line boundary, otherwise some vectors will cross cache lines. Performing SIMD on unaligned data incurs a performance penalty, or may fail completely for some older standards. In practice this only impacts the code used to allocate memory and define array variables: i) Use either `_mm_malloc` (Intel) or `memalign` (GNU), in place of default `malloc`;[9] and ii) append `_declspec(align(n))` to the usual array definition code.[10] For details on efficient use of cache and memory access, see Crago et al. (2018) and Geng et al. (2018).

2.4   Performance analysis

Not all applications require code optimisation and this must be assessed on a problem-specific basis. Code optimisation effort should be informed by performance analysis to ensure potential reductions in computation time are worth extra development time. It is common for this to be overlooked in practice (Hurn et al., 2016; Lee et al., 2010b).

Profiling and static code analysis software can be used to obtain performance statistics on timing and memory bottlenecks, e.g., Intel VTune Amplifier[11] and Intel Advisor.[12] These tools are also essential to investigate and control compiler output. In addition to profiling code, it is important to estimate the theoretical peak performance for your hardware. One can obtain a rough best-case runtime by comparing this peak performance with an estimate of the number of floating point operations your algorithm will perform, on average (Hurn et al., 2016).

---

[7]Access matrices according to the matrix storage format. For example, the row-major format is used in C/C++ (access row-by-row) and the column-major format is used in R (access column-by-column).
[8]The CPU-Z utility (https://www.cpuid.com) is a useful tool to identify cache line sizes.
[9]On most systems, `malloc` defaults are 8byte or 16byte.
[10]For example, `_declspec(align(64)) double A[100]` is a 64 byte aligned array.
[11]https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.
[12]https://software.intel.com/content/www/us/en/develop/tools/advisor.html.

Lastly, most algorithms require some sequential operations, and not all calculations can be efficiently mapped to SIMD or multithreading. Amdahl's law (Amdahl, 1967) states the speedup factor, s, is bounded by

$$s \leq \frac{C_S + C_P}{C_S + C_P/P}, \tag{2.1}$$

where C_P is the sequential runtime of code that can be parallelised, C_S is the runtime of code that must remain sequential, and P is the core count or vector length. Thus, $s = (C_S + C_P)/C_S$ is the maximum speedup as $P \to \infty$.

2.5   A note on random number generation

When combining statistics with parallel computing it is important to consider initialisation and usage of random number generators (RNGs). Dealing with these aspects poorly can lead to invalid results, even within standard environments like R. In particular, the use of different seed values (one per thread) for the same sequential RNG should be approached with caution. For example, sequences of Linear Congruential Generators can become correlated if a linear sequence of seeds is used (Davies and Brooks, 2007).

Broadly there are three approaches to deal with this: (i) generate all required randomness serially and distribute among parallel processes; (ii) use a random number generator that can be split into independent sub-streams, for example a generator that supports *Skip Ahead* or *Leapfrog* methods; or (iii) use a sequence of random number generators that are statistically independent for the same seed value. We utilise the last option through the `VSL_BRNG_MT2203` generator family (available within Intel MKL) that provides statistically independent Mersenne Twister generators. See Bradley et al. (2011) and Lee et al. (2010a) for details.

2.6   Summary

This introduction to SIMD, OpenMP, memory usage, code analysis, and RNGs is not specific to Bayesian statistics. However, the ideas are important to efficiently implement steps within the increasingly computationally expensive algorithms that form a core part of modern practical Bayesian techniques. In Sections 3 and 4 we demonstrate the application of these guidelines through a detailed tutorial and number of case studies in settings of direct interest to Bayesian practitioners.

3   A practical tutorial demonstration for R users

In this section, we provide a practical demonstration of the computational benefits of directly accessing CPU SIMD operations for ABC-based inference. We begin with R implementations, step through relevant optimisation within R, then demonstrate the computational advantages of direct SIMD access using C and OpenMP.

3.1 Prior predictive sampling for approximate Bayesian computation

ABC techniques are powerful for inference with intractable likelihood functions (Sisson et al., 2018) and are routinely used to study complex stochastic models (Ross et al., 2017; Tanaka et al., 2006). Given data, $\mathcal{D}_{\mathrm{obs}}$, the parameter prior density, $\pi(\boldsymbol{\theta})$, a discrepancy metric, ρ, and a vector of sufficient (or informative) summary statistics, $S(\mathcal{D}_{\mathrm{obs}})$, then ABC rejection sampling generates approximate posterior samples by first generating artificial data from the prior predictive distribution with density,

$$\pi(\mathcal{D}) = \int_{\boldsymbol{\Theta}} s(\mathcal{D}; \boldsymbol{\theta}) \pi(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}, \tag{3.1}$$

where $s(\mathcal{D}; \boldsymbol{\theta})$ is the probability density of the data generation process for a fixed parameter vector $\boldsymbol{\theta}$ in parameter space $\boldsymbol{\Theta}$. A small proportion of samples are accepted to form an approximation to the posterior,

$$\pi(\boldsymbol{\theta} \mid \mathcal{D}_{\mathrm{obs}}) \approx \pi(\boldsymbol{\theta} \mid \rho(S(\mathcal{D}_{\mathrm{obs}}), S(\mathcal{D})) \le \epsilon) = \frac{\mathbb{P}\left(\rho(S(\mathcal{D}_{\mathrm{obs}}), S(\mathcal{D})) \le \epsilon \mid \boldsymbol{\theta}\right) \pi(\boldsymbol{\theta})}{\pi(\mathcal{D}_{\mathrm{obs}})}$$

with a sufficiently small acceptance threshold ϵ. This process is shown in Algorithm 1.

Algorithm 1 ABC rejection sampling.

1: **repeat**
2:     Generate candidate sample $\boldsymbol{\theta}^* \sim \pi(\boldsymbol{\theta})$;
3:     Use stochastic simulation to generate prior predictive data $\mathcal{D}^* \sim s(\mathcal{D}; \boldsymbol{\theta}^*)$;
4: **until** $\rho(S(\mathcal{D}_{\mathrm{obs}}), S(\mathcal{D}^*)) \le \epsilon$;
5: Accept $\boldsymbol{\theta}^*$ as an approximate posterior sample;

ABC rejection sampling is rarely implemented in this direct manner (Algorithm 1 is serial, and produces a random number of candidate samples). Rather, it is common to generate a fixed number, N, of prior predictive joint samples, $(\boldsymbol{\theta}^*, \mathcal{D}^*)$. The acceptance threshold, ϵ, is then selected *a posteriori* based on the empirical distribution of the discrepancy metric, ρ. See e.g. Fan and Sisson (2018) and Warne et al. (2019) for detailed reviews of ABC methods. ABC-based Monte Carlo estimators converge slowly in *mean-square* (Barber et al., 2015), and require large N for reliable inference.

3.2 Example model: a genetic toggle switch

We consider a genetic toggle switch model (Bonassi et al., 2011). Let $u_i(t) \ge 1$ and $v_i(t) \ge 1$ represent expression levels of genes u and v at time t for cells $i = 1, 2 \ldots, C$. Gene expression evolves according to two coupled *stochastic differential equations* (SDEs)

$$\begin{aligned}
\mathrm{d}u_i(t) &= \left(\frac{\alpha_u}{1 + v_i(t)^{\beta_u}} - (1 + 0.03 u_i(t)) \right) \mathrm{d}t + \frac{1}{2} \mathrm{d}W_{i,u}(t), \\
\mathrm{d}v_i(t) &= \left(\frac{\alpha_v}{1 + u_i(t)^{\beta_v}} - (1 + 0.03 v_i(t)) \right) \mathrm{d}t + \frac{1}{2} \mathrm{d}W_{i,v}(t),
\end{aligned} \tag{3.2}$$

where parameters α_u, β_u, α_v and β_v define gene inhibition and $W_{i,u}(t)$ and $W_{i,v}(t)$ are independent Wiener processes. The observation process is

$$y_i = u_i(T) + \mu + \mu\sigma\frac{\eta_i}{u_i(T)^\gamma}, \quad i = 1, 2, \ldots, C, \tag{3.3}$$

where T is the observation time, μ, σ and γ control the error rate and η_i are standard normal random variables. The data are the measurements for all cells, $\{y_i\}_{i=1}^C$. Sampling the prior predictive distribution is needed to infer the parameters $\boldsymbol{\theta} = (\mu, \sigma, \gamma, \alpha_u, \alpha_v, \beta_u, \beta_v)'$ using ABC. We adopt independent uniform priors as chosen by Bonassi et al. (2011), $\mu \sim \mathcal{U}(250, 400)$, $\sigma \sim \mathcal{U}(0.05, 0.5)$, $\gamma \sim \mathcal{U}(0.05, 0.35)$, $\alpha_u, \alpha_v \sim \mathcal{U}(0, 50)$, and $\beta_u, \beta_v \sim \mathcal{U}(0, 7)$. We follow Vo et al. (2019) in adopting 19 equally spaced quantiles of the empirical distribution of $\{y_i\}_{i=1}^C$ as the vector of summary statistics.

3.3   Implementation and optimisation using R

To simulate the toggle switch system (Equation (3.2)) and observation process (Equation (3.3)) we can use the *Euler-Maruyama* scheme (Maruyama, 1955),

$$\begin{aligned}
u_i(t + \Delta t) &= u_i(t) + \Delta t\frac{\alpha_u}{1 + v_i(t)^{\beta_u}} - \Delta t(1 + 0.03u_i(t)) + 0.5\sqrt{\Delta t}\xi_{i,u}(t), \\
v_i(t + \Delta t) &= v_i(t) + \Delta t\frac{\alpha_v}{1 + u_i(t)^{\beta_v}} - \Delta t(1 + 0.03v_i(t)) + 0.5\sqrt{\Delta t}\xi_{i,v}(t),
\end{aligned} \tag{3.4}$$

where Δt is the discretisation step and $\xi_{i,u}(t)$ and $\xi_{i,v}(t)$ are independent normal random variables. For simplicity, we will assume $\Delta t = 1$. This can be implemented directly using R as shown in Listing 1.

```
1   simulate.tsw.SDE <- function(theta, T, C) {
2     y <- numeric(C); mu <- theta[1]; sigma <- theta[2]; gam <- theta[3]
3     alpha.u <- theta[4]; alpha.v <- theta[5]
4     bet.u <- theta[6]; bet.v <- theta[7]
5     for (i in 1:C) { # loop over cells
6       ut <- 10; vt <- 10
7       for (j in 2:T){ # evolve gene expression dynamics
8         p.u <- vt^bet.u; p.v <- ut^bet.v
9         ut <- 0.97*ut + alpha.u/(1+p.u) - 1.0 + 0.5*rnorm(1,0,1)
10        vt <- 0.97*vt + alpha.v/(1+p.v) - 1.0 + 0.5*rnorm(1,0,1)
11        ut <- ifelse(ut < 1.0,1.0,ut)
12        vt <- ifelse(vt < 1.0,1.0,vt)
13      }
14      # make noise observation
15      y[i] <- ut + mu + sigma*mu*rnorm(1,0,1)/(ut^gam)
16    }
17    y <- ifelse(y < 1.0, 1.0, y)
18    return(y)
19  }
```

Listing 1: Naïve R implementation of the toggle switch model.

Here, the pair of SDEs for $u_i(t)$ and $v_i(t)$ are evolved one cell at a time, for $i = 1, 2, \ldots, C$. Such a naïve implementation is typical for initial prototypes, but it is profoundly inefficient with a single simulation with $T = 600$ and $C = 8000$ taking 35 seconds. This is not practical for ABC sampling, even with individual simulations distributed across a HPC cluster.

One can achieve substantial computational improvements by re-writing the R code in terms of vector and matrix mathematics. This exploits optimised linear algebra libraries such as BLAS and LAPACK that often use SIMD. Listing 2 is an example of vectorised R code for the toggle switch model.

```
 1  simulate.tsw.SDE <- function(theta, T, C) {
 2    ut <- numeric(C); vt <- numeric(C)
 3    mu <- theta[1]; sigma <- theta[2]; gam <- theta[3]
 4    alpha.u <- theta[4]; alpha.v <- theta[5]
 5    bet.u <- theta[6]; bet.v <- theta[7]
 6    ut[1:C] <- 10; vt[1:C] <- 10
 7    zeta <- matrix(nrow=C,ncol=2*(T-1)+1)
 8    zeta[,] <- rnorm(C*(2*(T-1)+1),0,1) #generate all random variates
 9    for (j in 2:T) { # evolve all cells together with vectors
10      p.u <- vt^bet.u; p.v <- ut^bet.v
11      ut <- 0.97*ut + alpha.u/(1+p.u) - 1.0 + 0.5*zeta[1:C,2*(j-1)]
12      vt <- 0.97*vt + alpha.v/(1+p.v) - 1.0 + 0.5*zeta[1:C,2*(j-1) + 1]
13      ut[ut < 1.0] <- 1.0; vt[vt < 1.0] <- 1.0;
14    }
15    # make noise observation
16    y <- ut + mu+  sigma*mu*zeta[1:C,1]/(ut^gam)
17    y[y < 1.0] <- 1.0
18    return(y)
19  }
```

Listing 2: Optimised R implementation of the toggle switch model.

The key changes are: 1) genes of all cells are stored in vectors of length C; 2) all Gaussian random variates required for the entire simulation are generated in a single call of `rnorm()`; 3) all genes are simulated together using R's vector maths functions; and 4) logical indexing is used instead of `ifelse()` to implement the boundary condition. The improvements are substantial, taking 1.3 seconds to perform a single simulation with $T = 600$ and $C = 8000$. This is more than $25\times$ improvement, without any multithreading, explicit SIMD operations, or Rcpp code. See the excellent text by Gillespie and Lovelace (2017) for details.

The purpose of comparing Listings 1 and 2 is to highlight the importance of optimising serial code. Now that the R implementation is efficient, we can distribute the process across multiple cores using the `doParallel` package as shown in Listing 3. Distributing $N = 8064$ draws from the prior predictive across 16 cores (Intel Xeon E5-2680v3 processor[13]) takes just over 11 minutes!

---

[13]https://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2-50-GHz-.

```
1   library(tictoc)
2   library(doParallel)
3   source("simulateToggleSwitchSDEopt.R")
4   # set up problem size
5   T <- 600; C <- 8000; N <- 8064
6   # set up cluster
7   cl <- makeCluster(24)
8   registerDoParallel(cl)
9   tic() # Generate prior predictive samples
10  obs_vals <- foreach(k = 1:N) %dopar%  {
11    # sample the prior
12    theta <- runif(7,c(250.0,0.05,0.05,0.0,0.0,0.0,0.0),
13                     c(400.0,0.5,0.35,50.0,50.0,7.0,7.0))
14    # run model simulation
15    c(theta,simulate.tsw.SDE(theta,T,C))
16  }
17  toc() # report timing
18  stopCluster(cl) # clean up
```

Listing 3: Multithreaded prior predictive sampling using the R package `doParallel`.

For some applications, optimised R can be fast enough and we do not advocate the use of further optimisation for all Bayesian applications (See Section 2.4). However, the trade-off between development time and runtime is very application specific. For this example, even using 16 cores, the optimised R code will take around 24 hours to draw $N = 1,000,000$ prior predictive samples, which may be insufficient for ABC inference with small ϵ. To motivate progressing beyond R, note that the indirect access of SIMD through optimised BLAS and LAPACK libraries has limitations in terms of cache utilisation for this application. This is highlighted in the next section.

3.4   Optimisation using C and SIMD operations

Direct access to SIMD through C with OpenMP enables superior memory access patterns that ensure substantially higher utilisation of the VPUs (see Section 5 for alternatives). While the optimised C implementations presented here are more complex than the optimised R implementation (Listing 2), they provide a practical example that will help reduce the learning curve for practitioners.

Listing 4 provides the main structure of the C program. Note the inclusion of the header files for the MKL libraries and the OpenMP library; we will point out calls to these libraries as necessary. Definitions of the constants `VECL` and `ALIGN` refer to the width of the SIMD vectors and the memory alignment to ensure vectors never cross cache lines (see Section 2). The functions `simulate_tsw_SDE` and `main` represent the C equivalents of the R code Listings 2 and 3. C implementations are given in Listings 5 and 6, the details of which are discussed below.

```
1  /* standard C headers*/
2  #include <stdio.h>
3  #include <math.h>
4  #include <string.h>
5  /* Intel headers */
6  #include <mkl.h>
7  #include <mkl_vsl.h>
8  /* OpenMP header */
9  #include <omp.h>
10
11 #define VECL 4    /* for AVX2 (256 bit = 4 doubles) */
12 #define ALIGN 64 /* 64 byte cache lines */
13
14 void
15 simulate_tsw_SDE(VSLStreamStatePtr stream,
16                  double * restrict theta, int T, int C,
17                  double * restrict zeta, double * restrict y)
18   /* ... code for model simulation ... */
19 }
20
21 int
22 main(int argc,char **argv) {
23   /* ... code for parallel prior predictive sampling ... */
24 }
```

Listing 4: Overall structure of the optimised C implementation.

The optimised C implementation of the toggle switch model is given in Listing 5. From an algorithmic perspective this code can be considered a hybrid between the two R implementations (Listings 1 and 2). That is, cells are evolved in small blocks of length VECL, as opposed to individually (Listing 1) or all together (Listing 2). The outer loop iterates over the cell index c in strides of VECL (the E5-2680v3 CPU supports 265 bit vectors, giving VECL = 4 doubles). As in the optimised R code (Listing 2), all the random variates required within each block are pre-computed using the vdRngGaussian function from MKL. In each block there are VECL noisy observations, y[c],y[c+1],...,y[c+VECL-1], to be computed (one per cell). Using SIMD we can evolve the SDE pair associated with the y[c] observation and obtain the others in the same block at the same time. This is done with a second loop over the index c2 that represents the position in the SIMD vector. Note the use of the aligned statement within the simd directive. This allows the compiler to assume the arrays zeta (random variates) and y (simulated data) are aligned to the cache boundary and enables more efficient machine code to be generated. The SIMD loop is not really iterating over values of c2, but rather synchronously executing each statement within the loop for all values of c2 concurrently. This leads to exceptional reuse of L1 cache: since the SDE state variable u_t and v_t are updated in-place, all computation is performed using the fast CPU memory.

```
1   void
2   simulate_tsw_SDE(VSLStreamStatePtr s,
3                    double * restrict theta, int T, int C,
4                    double * restrict zeta, double * restrict y) {
5     double mu = theta[0], sigma = theta[1], gamma = theta[2];
6     double *alpha = theta+3, *beta = theta+5;
7     /* process cells in blocks of VECL*/
8     for (int c=0;c<C;c+=VECL){
9       /* Generate all the random variates for these realisations*/
10      vdRngGaussian(VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2,s,2*VECL*T,
11              zeta,0.0,1.0);
12      /* simulate all trajectories for this block in SIMD*/
13      #pragma omp simd aligned(zeta:ALIGN, y:ALIGN)
14      for (int c2=0;c2<VECL;c2++) {
15        /* copy parameters to ensure in cache/registers*/
16        double _gamma = gamma, _sigma = sigma, _mu = mu;
17        double alpha_u = alpha[0], alpha_v = alpha[1];
18        double beta_u = beta[0],  beta_v = beta[1];
19        double u_t = 10, v_t = 10;
20        /* evolve u/v pairs for this cell  */
21        for (int j=1;j<T;j++) {
22          double p_u = pow(v_t,beta_u);
23          double p_v = pow(u_t,beta_v);
24          u_t *= 0.97; u_t += alpha_u/(1.0 + p_u) - 1.0;
25          v_t *= 0.97; v_t += alpha_v/(1.0 + p_v) - 1.0;
26          double zeta_u = zeta[(j-1)*VECL*2 + c2];
27          double zeta_v = zeta[(j-1)*VECL*2 + 4 + c2];
28          u_t += 0.5*zeta_u; u_t  = (u_t >= 1.0) ? u_t : 1.0;
29          v_t += 0.5*zeta_v; v_t  = (v_t >= 1.0) ? v_t : 1.0;
30        }
31        /* make noisy observation */
32        y[c+c2] = u_t + _mu +
33                 _sigma*_mu*zeta[(T-1)*VECL*2+c2]/pow(u_t,_gamma);
34        y[c+c2] = (y[c+c2] >= 1.0) ? y[c+c2]: 1.0;
35      }
36    }
37  }
```

Listing 5: Optimised C implementation of the toggle switch model. OpenMP is used to execute the innermost loop using SIMD. MKL is used for RNGs.

It is possible to do even better, since computing all Gaussian random variates for each block represents a trade-off. To understand this, note that advanced interfaces, such as compiler intrinsic functions (see discussion in Section 5), can enable efficient RNG sampling within the rest of the Euler-Maruyama update. However, this cannot be done with OpenMP as the function vdRngGaussian has overheads that dominate for very small sample sizes. Conversely, generating all the Gaussian random variates at the start with a single vdRngGaussian call forces the CPU to access L2 cache after a few blocks have been processed, resulting in loss of performance. This trade-off to memory access presented in Listing 5 cannot be efficiently replicated in R. Using BLAS and LAPACK functions for small vectors results in similar performance issues as for the vdRngGaussian function. Therefore, the best that can be done with R is to use long vectors to evolve all cells together.

```c
int
main(int argc,char **argv) {
  /* set up problem size*/
  int T = 600, C = 8000, K = 7, N = 8064, seed = 1337;
  /*allocate aligned memory for generated data */
  double *obs_vals = (double *)_mm_malloc(C*sN*sizeof(double),ALIGN);
  /* allocate aligned memory prior samples */
  double *theta = (double *)_mm_malloc(K*N*sizeof(double),ALIGN);
  /* compute simulations in parallel*/
  #pragma omp parallel shared(seed,N,C,obs_vals,theta)
  {
    VSLStreamStatePtr s;
    /* get thread information and assign workload */
    int tid = omp_get_thread_num();
    int N_per_thr = N/omp_get_num_threads();
    /* initialise RNG stream for this thread */
    vslNewStream(&s,VSL_BRNG_MT2203+tid,seed);
    /*allocate aligned memory for Gaussian random variates */
    double *zeta = (double *)_mm_malloc(2*VECL*T*sizeof(double),ALIGN);
    /* compute simulations in this threads workload */
    for (int k=tid*N_per_thr;k<(tid+1)*N_per_thr;k++) {
      /* pointers to memory for this sample*/
      double *p = theta + k*7, *D = obs_vals + k*C;
      /* sample prior */
      vdRngUniform(VSL_RNG_METHOD_UNIFORM_STD,s,1,p,250.0,400.0);
      vdRngUniform(VSL_RNG_METHOD_UNIFORM_STD,s,1,p+1,0.05,0.5);
      vdRngUniform(VSL_RNG_METHOD_UNIFORM_STD,s,1,p+2,0.05,0.35);
      vdRngUniform(VSL_RNG_METHOD_UNIFORM_STD,s,2,p+3,0.0,50.0);
      vdRngUniform(VSL_RNG_METHOD_UNIFORM_STD,s,2,p+5,0.0,7.0);
      /* run simulation and store observations */
      simulate_tsw_SDE(s,p,T,C,zeta,D);
    }
    /* clean up memory */
    vslDeleteStream(&s);
    _mm_free(zeta);
  }
  /* \ldots code to output results to file goes here ... */
  exit(0);
}
```

Listing 6: Multithreaded prior predictive sampling using C and OpenMP threading

Armed with a highly efficient SIMD implementation of the model, we can generate prior predictive samples across multiple CPU cores. This uses the multithreading pre-processor directive, `#pragma omp parallel`, as shown in Listing 6. Each thread is provided with its own random number stream using the Intel MKL/VSL function `vslNewStream` with independence guarenteed through the VSL_BRNG_MT2203 generator family (see Section 2). The memory allocated to store prior samples, `theta`, and simulation outputs, `obs_vals`, are shared across threads, using the `shared` clause, and thread specific offsets are computed, using the OpenMP functions `omp_get_thread_num()` and `omp_get_num_threads()`. The loop over index `k` is performing the prior predictive sampling tasks assigned to each thread. Unlike the R implementation (Listing 3), we explicitly partition the work amongst the available threads to allow reuse of the memory

allocated for the Gaussian random variates, `zeta`. OpenMP does support various forms of automated scheduling of thread tasks, but for this example explicit partitioning is more straightforward.

3.5   Benchmarks

We compare the R implementations (Listings 1, 2 and 3) with the C implementation (Listings 4, 5 and 6) in terms of the time taken to generate a fixed number, N, of prior predictive samples given P cores. The fastest sampler would, in turn, produce a smaller Monte Carlo error for a fixed computational budget. For example, the standard central limit theorem suggests a $4\times$ speedup will result in roughly $1/2$ the Monte Carlo error for the same computational budget. In an ABC setting there is also bias to consider, so reducing the model simulation time is absolutely crucial since the acceptance threshold ϵ must be sufficiently small.

The codes are benchmarked using two Intel Xeon E5-2680v3 (Haswell) processors or a single Intel Xeon Gold 6140 (Skylake) processor. The E5-2680v3 supports the AVX2 instruction set (256 bit vector) and the 6140 supports the AVX512 instruction set (512 bit vectors). Both processor architectures have similar serial performance with cores clocked at around 2.5 GHz. We benchmark with different numbers of cores, $P = 1, 2, 4, 8, 16$ for each implementation. For each value of P the benchmark is repeated four times, to obtain mean computation times $\hat{C}_T$, with each replicate generating $N = 8064$ prior predictive samples. For fairness, we compiled R (version 3.3.1) from source using the Intel compiler suite and linked against MKL for optimised BLAS and LAPACK routines. The Intel C compiler version is 17.0.1 (compatible with the GNU C complier version 6.3.0). To demonstrate the difference between speed-up obtained from the improved memory utilisation and the usage of SIMD, we also compile a version of the C implementation with SIMD disabled using the `-no-vec` compiler option.

Figure 3 summarises the results in terms of runtimes $\hat{C}_T$ (Figure 3(a),(c),(e)) and speed-up factor $\hat{C}_T/\hat{C}_B$ for various serial baselines $\hat{C}_B$ (Figure 3(b),(d),(f)). See the Supplementary Material (Warne et al., 2021) for precise data tables of runtimes. Using the naïve R code as the baseline (Figure 3(a)–(b)), a speedup of $16\times$ is possible with $P = 16$ as one would expect. However, this pales by comparison with the more than $430\times$ speedup using the optimised R code and $P = 16$, demonstrating the value in optimising serial code before implementing parallelism. The improvement is even greater for the C code with speedups of more than $1000\times$ (no vectors, $P = 16$), $3800\times$ (AVX2 vectors, $P = 16$), and $4300\times$ (AVX512, $P = 16$). These impressive numbers are against a naïve baseline and more sensible numbers are obtained using the optimised R code as a baseline (Figure 3(c)–(d)). Here optimised C code can achieve $130\times$ speedup (AVX2, $P = 16$) and $150\times$ speedup (AVX512, $P = 16$) compared with only $16\times$ speedup with optimised R code and $P = 16$. Finally, even with the C code as a baseline (Figure 3(e)–(f)), SIMD enables $30\times$ speedup (AVX2, $P = 16$) and almost $40\times$ speedup (AVX512, $P = 16$). In this case, the speedup from multithreading alone is only $13\times$ for $P = 16$. There are a few reasons that could cause this, but it is likely that each thread does not perform enough work to mask threading overheads. $P \leq 4$, seems to provide a large enough workload per thread leading to $15\times$ speedup (AVX2, $P = 4$) and $20\times$ speedup (AVX512, $P = 4$).
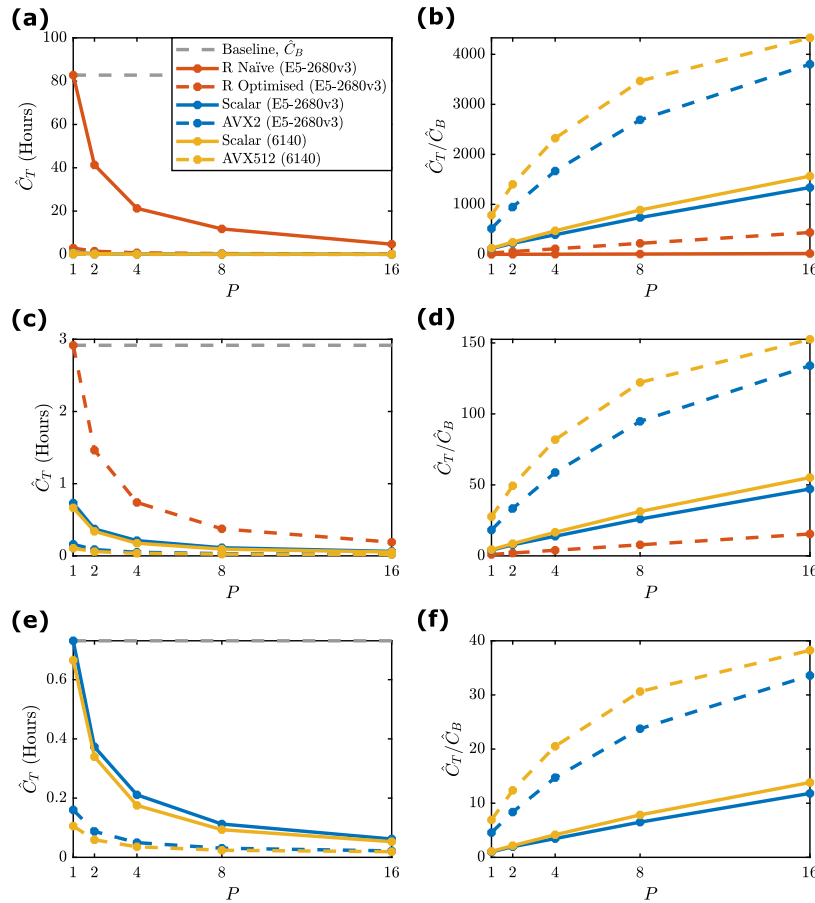
Figure 3: Benchmarking results for the prior predictive sampling application. (a) Mean runtimes, $\hat{C}_T$, using P cores for various implementations: R Naïve (Xeon E5-2680v3; solid red), R optimised (Xeon E5-2680v3; dashed red), C with scalar operations only (Xeon E5-2680v3; blue solid), C with AVX2 vectors (Xeon E5-2680v3; blue dashed), C with scalar operations only (Xeon Gold 6140; yellow solid), and C with AVX512 vectors (Xeon Gold 6140; yellow dashed). (b) Computational improvements for each optimisation relative to baseline $\hat{C}_B$ (R Naïve with $P = 1$). (c) and (d) same as (a) and (b) but using a different baseline $\hat{C}_B$ (R Optimised with $P = 1$). (e) and (f) same as (a) and (b) but using a different baseline $\hat{C}_B$ (C scalar with $P = 1$ on Xeon E5-2680v3).

3.6 Summary

This tutorial has demonstrated the computational benefits of direct access to SIMD operations in comparison to indirect access through pre-compiled BLAS and LAPACK libraries available in languages such as R and Matlab. Our C implementation (Listings 5

and 6) is more that $20\times$ faster than our optimised R implementation (Listing 2 and 3) for the same CPU hardware and number of available cores, P. Restricting comparison to the C implementation, SIMD operations can improve the performance of prior predictive sampling for ABC by a factor of more than $6\times$ compared with standard scalar implementations. Given the enormous number of prior predictive samples required to obtain meaningful parameter estimates with ABC (Barber et al., 2015), we submit that these techniques are highly relevant for practitioners dealing with such applications. All codes presented here are available as Supplementary Material (Warne et al., 2021).

4   Case studies

In this section, we explore the benefit of SIMD operations in combination with multi-threading using two case studies relevant to Bayesian practitioners.

4.1   Case study 1: weakly informative priors

Here, we consider the selection of weakly informative priors (Gelman, 2006) within the framework proposed by Evans and Jang (2011) using the implementation by Nott et al. (2018). Following Nott et al. (2018), the prior predictive p-value,

$$p(\mathcal{D}_{\text{obs}}) = \mathbb{P}\left(\frac{1}{\pi(\mathcal{D})} \geq \frac{1}{\pi(\mathcal{D}_{\text{obs}})}\right) = \mathbb{P}\left(\pi(\mathcal{D}) \leq \pi(\mathcal{D}_{\text{obs}})\right), \tag{4.1}$$

is a measure of Bayesian model criticism, where $\mathcal{D}_{\text{obs}}$ are the observational data, and $\pi(\mathcal{D})$ is the prior predictive distribution (Equation (3.1)). Equation (4.1) provides a p-value for prior-data conflict (Evans and Jang, 2011) associated with low prior density assigned to parameters with good model fit. If γ is a pre-defined small cut-off, then $p(\mathcal{D}_{\text{obs}}) \leq \gamma$ signifies prior-data conflict.

Consider a family of priors, $\pi(\boldsymbol{\theta} \mid \boldsymbol{\lambda})$ with the hyperparameter $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$. A base prior, $\pi(\boldsymbol{\theta} \mid \boldsymbol{\lambda}_0)$ for $\boldsymbol{\lambda}_0 \in \boldsymbol{\Lambda}$, represents the current best knowledge of the parameter $\boldsymbol{\theta}$. Let

$$\pi(\mathcal{D} \mid \boldsymbol{\lambda}) = \int_{\boldsymbol{\Theta}} \mathcal{L}(\boldsymbol{\theta}; \mathcal{D})\pi(\boldsymbol{\theta} \mid \boldsymbol{\lambda})\, \mathrm{d}\boldsymbol{\theta} \tag{4.2}$$

be the prior predictive density for the prior $\pi(\boldsymbol{\theta} \mid \boldsymbol{\lambda})$ and likelihood $\mathcal{L}(\boldsymbol{\theta}; \mathcal{D})$, and let $p_{\boldsymbol{\lambda}}(\mathcal{D}_{\text{obs}})$ be the prior predictive p-value under $\pi(\mathcal{D} \mid \boldsymbol{\lambda})$ (equivalent to Equation (4.1)),

$$p_{\boldsymbol{\lambda}}(\mathcal{D}_{\text{obs}}) = \mathbb{P}\left(\pi(\mathcal{D} \mid \boldsymbol{\lambda}) \leq \pi(\mathcal{D}_{\text{obs}} \mid \boldsymbol{\lambda})\right). \tag{4.3}$$

Assume data are generated under the base prior predictive distribution, that is, $\mathcal{D}_0 \sim \pi(\mathcal{D} \mid \boldsymbol{\lambda}_0)$. The task is to find values of $\boldsymbol{\lambda}$ such that,

$$\mathbb{P}\left(p_{\boldsymbol{\lambda}}(\mathcal{D}_0) \leq \gamma\right) < \mathbb{P}\left(p_{\boldsymbol{\lambda}_0}(\mathcal{D}_0) \leq \gamma\right). \tag{4.4}$$

Priors satisfying Equation (4.4) are weakly informative with respect to the base prior. Weak informativity indicates less prior-data conflict than the base. Equivalently, weak

informativity at level α indicates $\gamma_{\boldsymbol{\lambda},\alpha} > \gamma_{\boldsymbol{\lambda}_0,\alpha}$ where $\gamma_{\boldsymbol{\lambda},\alpha}$ and $\gamma_{\boldsymbol{\lambda}_0,\alpha}$ are the prior-data conflict cut-offs such that $\mathbb{P}\left(p_{\boldsymbol{\lambda}}(\mathcal{D}_0) \leq \gamma_{\boldsymbol{\lambda},\alpha}\right) = \mathbb{P}\left(p_{\boldsymbol{\lambda}_0}(\mathcal{D}_0) \leq \gamma_{\boldsymbol{\lambda}_0,\alpha}\right) = \alpha$.

Given a set of K hyperparameter values, $\boldsymbol{\lambda}_1, \ldots, \boldsymbol{\lambda}_K \in \boldsymbol{\Lambda}$, with $\boldsymbol{\lambda}_0$ corresponding to the base prior, the task is to compute $\gamma_{\boldsymbol{\lambda}_k,\alpha}$ such that $\mathbb{P}\left(p_{\boldsymbol{\lambda}_k}(\mathcal{D}_0) \leq \gamma_{\boldsymbol{\lambda}_k,\alpha}\right) = \alpha$ for $k = 0, 1, \ldots, K$ and $\mathcal{D}_0 \sim \pi(\mathcal{D} \mid \boldsymbol{\lambda}_0)$. The process proceeds as in Algorithm 2. The output of Algorithm 2 is a set of α level cutoffs $\{\gamma_{\boldsymbol{\lambda}_0,\alpha}, \gamma_{\boldsymbol{\lambda}_1,\alpha}, \ldots, \gamma_{\boldsymbol{\lambda}_K,\alpha}\}$, to derive a set of weakly informative prior distributions with respect to the base prior $\pi(\boldsymbol{\theta} \mid \boldsymbol{\lambda}_0)$, that is, $\{\pi(\boldsymbol{\theta} \mid \boldsymbol{\lambda}_k) : \gamma_{\boldsymbol{\lambda}_k,\alpha} > \gamma_{\boldsymbol{\lambda}_0,\alpha}, k = 1, 2, \ldots, K\}$.

Algorithm 2 Weak informativity test.

1: Initialise hyperparameters $\boldsymbol{\lambda}_0, \boldsymbol{\lambda}_1, \ldots, \boldsymbol{\lambda}_K \in \boldsymbol{\Lambda}$, with base parameter $\boldsymbol{\lambda}_0$;
2: **for** $k \in [0, 1, \ldots, K]$ **do**
3:     **for** $i \in [1, 2, \ldots, N]$ **do**
4:         Generate data from base prior predictive $\mathcal{D}_0^{(i)} \sim \pi(\mathcal{D} \mid \boldsymbol{\lambda}_0)$;
5:         Generate data from prior predictive $\mathcal{D}_k^{(i)} \sim \pi(\mathcal{D} \mid \boldsymbol{\lambda}_k)$;
6:         Evaluate $\pi_0^i \leftarrow \pi(\mathcal{D}_0^{(i)} \mid \boldsymbol{\lambda}_k)$ and $\pi_k^i \leftarrow \pi(\mathcal{D}_k^{(i)} \mid \boldsymbol{\lambda}_k)$;
7:     **end for**
8:     **for** $i \in [1, 2, \ldots, N]$ **do**
9:         Estimate p-value samples $p_k^i \leftarrow p_{\boldsymbol{\lambda}_k}(\mathcal{D}_0^{(i)}) \approx \frac{1}{N} \sum_{j=1}^{N} \mathbb{1}_{[0,\pi_0^i]}\left(\pi_k^j\right)$;
10:     **end for**
11:     Compute $\gamma_{\boldsymbol{\lambda}_k,\alpha}$ as the α quantile of $\{p_k^1, p_k^2, \ldots, p_k^N\}$;
12: **end for**

This process is extremely computationally intensive since the evidence term, $\pi(\mathcal{D} \mid \boldsymbol{\lambda})$, must be evaluated for many different $\mathcal{D}$. Adaptive sequential Monte Carlo (SMC) sampling (Supplementary Material, Warne et al., 2021) using likelihood annealing and Markov chain Monte Carlo (MCMC) proposals (Beskos et al., 2016; Chopin, 2002) is used to estimate $\pi(\mathcal{D} \mid \boldsymbol{\lambda})$. Algorithm 2 requires $2KN$ executions of the adaptive SMC sampler (Supplementary Material, Warne et al., 2021). Algorithm 2 represents a direct implementation of the weak informativity test for the purposes of demonstrating SIMD optimisation. However, other algorithmic optimisations could also be made. For example, it would be possible to re-use the N datasets generated by the base prior for each $k = 0, 1, \ldots, K$ and there may be computational benefits for doing so, such as variance reduction and likelihood evaluation reuse.

We consider weakly informative priors for the analysis of an acute toxicity test in which M groups of animals are given different dosages of some toxin, and the number of deaths in each group are recorded (Evans and Jang, 2011; Nott et al., 2018). A logistic regression model is applied for the number of deaths y_i in group $i \in [1, 2, \ldots M]$,

$$y_i \sim \text{Bin}\left(n_i, \frac{1}{1 + e^{\beta_0 + \beta_1 x_i}}\right), \quad i = 1, 2, \ldots, M, \tag{4.5}$$

where n_i and x_i are respectively the number of animals and the toxin dose level in the ith group of the experiment, and $\boldsymbol{\theta} = (\beta_0, \beta_1)'$ are the regression parameters. Given

data $\mathcal{D}_{\mathrm{obs}} = [y_{\mathrm{obs},1}, y_{\mathrm{obs},2}, \ldots, y_{\mathrm{obs},M}]$, the likelihood function is

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_{\mathrm{obs}}) = \prod_{i=1}^{M} \binom{n_i}{y_{\mathrm{obs},i}} \left(\frac{1}{1 + e^{\beta_0 + \beta_1 x_i}} \right)^{y_{\mathrm{obs},i}} \left(1 - \frac{1}{1 + e^{\beta_0 + \beta_1 x_i}} \right)^{n_i - y_{\mathrm{obs},i}}. \quad (4.6)$$

From the data in Evans and Jang (2011) we have $M = 4$, $n_1 = n_2 = n_3 = n_4 = 5$, $x_1 = -0.86$, $x_2 = -0.3$, $x_3 = -0.05$ and $x_4 = -0.75$. We adopt bivariate Gaussian priors from Nott et al. (2018), $\boldsymbol{\theta} \sim \mathcal{N}(\mathbf{0}, \mathrm{diag}(\boldsymbol{\lambda})^2)$ and $\boldsymbol{\lambda}_0 = (10, 2.5)'$.

Parallelisation and SIMD opportunities

Parallel implementations of SMC samplers using multithreading require thread synchronisation for both the resampling and annealing steps (Hurn et al., 2016; Lee et al., 2010a; Murray et al., 2016). Thus, it is more beneficial to distribute the K hyperparameter values in Algorithm 2 across P cores, with each thread computing the α quantile for K/P hyperparameters, since these may be performed independently. For every hyperparameter, we sequentially process the N p-value computations. The SMC sampler (Supplementary Material, Warne et al., 2021) can be accelerated through SIMD. SMC samplers are well suited to SIMD since sychronisation is maintained automatically. While there are many aspects of this sampler that use SIMD in the code example, we focus here on SIMD for the MCMC proposal kernel for diversification of particles as it has the greatest effect on performance.

The strategy is similar to the C implementation of the toggle switch model (Section 3). At SMC step n, each particle is perturbed via R_n MCMC steps. The same operations are performed at each MCMC iteration, with the exception of a single branch operation that arises from the accept/reject step. We process the particle updates in blocks of length V and evolve the R_n MCMC steps for this block together using SIMD. All of the Gaussian and uniform random variates required for the block are generated together before the block is processed. The SIMD MCMC proposals are performed as in Algorithm 3. Hadamard notation indicates element-wise division, $x \oslash y$, multiplication, $x \circ y$ and exponentiation, $x^{\circ a}$, for scalar a. Element-wise application of a function, f, over length V vectors is denoted by $f^V(x_{i:i+V}) = [f(x_i), f(x_{i+1}), \ldots, f(x_{i+V})]$. In practice, Algorithm 3 operates with the likelihood and prior on the log scale to avoid numerical underflow. For optimal performance, SIMD forms for the likelihood $\mathcal{L}^V$, prior density π^V and Gaussian proposal density ϕ^V are required. This is available for the logistic regression model, Gaussian priors and proposals used here.

Other aspects of the SMC sampler that can utilise SIMD include likelihood evaluations and the computation of effective sample sizes and weight updates (Supplementary Material, Warne et al., 2021). The scalar bottleneck is the multinomial resampling step that utilises the look-up method. Parallel approximations for resampling have been proposed (Murray et al., 2016) and demonstrated to be very effective for large scale SMC samplers. Since we focus here on a SIMD implementation of SMC, we do not implement this, however, extending Murray et al. (2016) to SIMD is an important piece of future work.

Algorithm 3 SIMD implementation of MCMC proposals for SMC.

1: **for** $i = 1, 1 + V, 1 + 2V, \ldots, N_p + 1 - V, N_p$ **do**
2:      Generate increments $\xi_{1:V}^{1:R_n} \sim \mathcal{N}(\mathbf{0}, h\hat{\Sigma})$;
3:      Generate uniform variates $u_{1:V}^{1:R_n} \sim \mathcal{U}(0, 1)$;
4:      **for** $r = 1, 2, \ldots, R_n$ **do**
5:          Generate proposal $\boldsymbol{\theta}_{1:V}^* \leftarrow \boldsymbol{\theta}_n^{(i:i+V)} + \xi_{1:V}^r$;
6:          $\alpha_{1:V} \leftarrow \mathcal{L}^V(\boldsymbol{\theta}_{1:V}^*; \mathcal{D})^{\circ t_n} \circ \pi^V(\boldsymbol{\theta}_{1:V}^* | \boldsymbol{\lambda}) \circ \phi^V(\boldsymbol{\theta}_n^{(i:i+V)}; \boldsymbol{\theta}_{1:V}^*, h\hat{\Sigma})$;
7:          $\alpha_{1:V} \leftarrow \alpha_{1:V} \oslash \Big[\mathcal{L}^V(\boldsymbol{\theta}_n^{(i:i+V)}; \mathcal{D})^{\circ t_n} \circ \pi^V(\boldsymbol{\theta}_n^{(i:i+V)} | \boldsymbol{\lambda}) \circ \phi^V(\boldsymbol{\theta}_{(i:i+V)}^*; \boldsymbol{\theta}_n^{(i:i+V)}, h\hat{\Sigma}) \Big]$;

8:          **for** $j = 1, 2, \ldots, V$ **do**
9:              **if** $u_j^r \leq \alpha_j$ **then**
10:                 $\boldsymbol{\theta}_n^{(i:i+j-1)} \leftarrow \boldsymbol{\theta}_j^*$;
11:             **end if**
12:         **end for**
13:     **end for**
14: **end for**

Performance

We test the performance improvement obtained through vectorisation and multithreading using the Xeon E5-2680v3 and Xeon Gold 6140 processors. The evaluation of the weak informativity test in Algorithm 2 is performed for $K = 400$ hyperparameters and $N = 400$ datasets. In all simulations, the K hyperparameter values are generated using a bivariate uniform distribution $\boldsymbol{\lambda}_k \sim \mathcal{U}([0.1, 10] \times [0.1, 20])$ for $k = 1, 2, \ldots, K$. The SMC sampling is performed with $N_p = 500$ particles to enable computation to remain largely within L1 and L2 cache. Results are provided in Figure 4 (see Supplementary Material for data tables, Warne et al., 2021).

Figure 4(b) shows almost perfect speedup from multithreading and consistent improvement due to SIMD when comparing scalar and SIMD performance for the same CPU and core count ($2.1\times$ for AVX2 and $2.3\times$ for AVX512). Diminishing returns are observed when stepping from 256 bit vector operations to 512 bit SIMD. In our bioassay example, the MCMC proposal kernel performs a small number of steps (usually no more than $R_n = 30$) and the number of particles in the SMC sampler is $N_p = 500$. The performance will improve for SIMD in cases where longer MCMC runs are required, since the MCMC step will dominate SMC iterations with more reuse of L1 cache.

Summary

We have presented the more challenging problem of weak informativity tests over a family of priors $\pi(\boldsymbol{\theta} \mid \boldsymbol{\lambda})$ with respect to a base prior $\pi(\boldsymbol{\theta} \mid \boldsymbol{\lambda}_0)$. This requires a large number of approximations of the posterior normalising constant for different datasets from the prior predictive distribution. The traditional approach of parallelisation of each SMC iteration would reduce the level of parallelism available across hyperparameters. Using the SIMD implementation of SMC we double our computational performance and
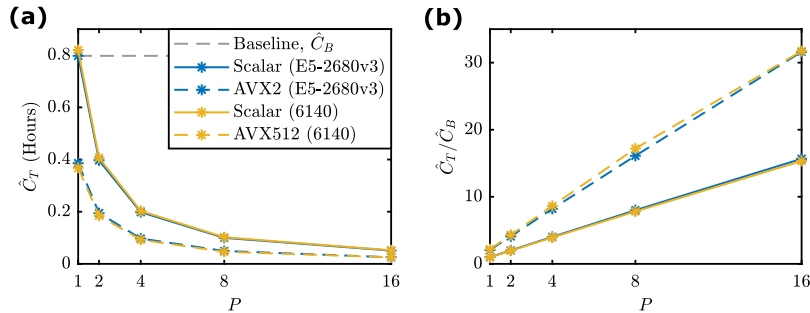
(a)

(b)

Figure 4: Benchmarking results for weak informativity test. (a) Mean runtimes, $\hat{C}_T$, using P cores for various implementations: C with scalar operations only (Xeon E5-2680v3; blue solid), C with AVX2 vectors (Xeon E5-2680v3; blue dashed), C with scalar operations only (Xeon Gold 6140; yellow solid), and C with AVX512 vectors (Xeon Gold 6140; yellow dashed). (b) Computational improvements for each optimisation relative to baseline $\hat{C}_B$ (C with scalar operations only with $P = 1$ on Xeon E5-2680v3).

can reserve multithreading across hyperparameters. A combination of parallelism and SIMD could be implemented to improve the performance of a single SMC step. However, this offers little benefit here since utilising threads for SMC forces hyperparameters to be processed serially. If HPC resources are available then hyperparameters could be distributed across servers: leaving both threading and SIMD for SMC steps.

4.2 Case study 2: parameter inference for a non-Gaussian asymmetric volatility model

We now consider an adaptive SMC sampler to perform parameter inference in eleven dimensions for the "bad environment – good environment" (BEGE) model of innovations on stock market returns (Bekaert et al., 2015; South et al., 2019).

The BEGE model

The BEGE model (Bekaert et al., 2015) is a non-Gaussian generalisation of the Glosten-Jagannathan-Runkle (GJR) asymmetric volatility model (Glosten et al., 1993) and is a generalised autoregressive conditional heteroskedasticity (GARCH) model (Bollerslev, 1986). The BEGE model describes the time-series of stock market returns, $\{r_t\}_{t \geq 0}$ using a model on innovation on returns, $\{u_t\}_{t \geq 0}$, that consists of a linear combination of positive "good environment" shocks, $\{\omega_{p,t}\}_{t \geq 0}$, and negative "bad environment" shocks, $\{\omega_{n,t}\}_{t \geq 0}$. The BEGE time-series evolves according to

$$r_{t+1} = u_{t+1} + \mu, \quad u_{t+1} = \sigma_p \omega_{p,t+1} - \sigma_n \omega_{n,t+1},$$
$$\omega_{p,t+1} \sim \tilde{\Gamma}(p_t, 1), \quad \omega_{n,t+1} \sim \tilde{\Gamma}(n_t, 1), \tag{4.7}$$

where μ is the conditional mean of returns, $\tilde{\Gamma}(k, 1)$ is the centered (de-meaned) gamma distribution with shape k and unit scale, $\{p_t\}_{t\geq 0}$ and $\{n_t\}_{t\geq 0}$ are respectively the shapes of the positive and negative shocks, and σ_p and σ_n are their constant scales. The shape parameters evolve according to

$$
\begin{aligned}
p_t &= p_0 + \rho_p p_{t-1} + \frac{\phi_p^+}{2\sigma_p^2} u_t^2 \mathbb{1}_{[0,\infty)} (u_t) + \frac{\phi_p^-}{2\sigma_p^2} u_t^2 \mathbb{1}_{(-\infty,0)} (u_t), \\
n_t &= n_0 + \rho_n n_{t-1} + \frac{\phi_n^+}{2\sigma_n^2} u_t^2 \mathbb{1}_{[0,\infty)} (u_t) + \frac{\phi_n^-}{2\sigma_n^2} u_t^2 \mathbb{1}_{(-\infty,0)} (u_t),
\end{aligned}
\tag{4.8}
$$

where p_0, n_0 are initial conditions, and ρ_p, ρ_n, ϕ_p^+, ϕ_n^+, ϕ_p^-, and ϕ_n^- are autoregression parameters.

We use S&P Composite Index returns over the period July 1926 to January 2018 (obtained from the Center of Research in Security Prices) consisting of $T = 1099$ months of logged monthly divided-adjusted returns, $\mathcal{D} = R_{\text{obs},T} = \{r_{\text{obs},t}\}_{0\leq t\leq T}$. Using these data, Bayesian inference on the unknown parameters $\boldsymbol{\theta} = (p_0, \sigma_p, \rho_p, \phi_p^+, \phi_p^-, n_0, \sigma_n, \rho_n, \phi_n^+, \phi_n^-, \mu)$ is performed. Priors are adopted from South et al. (2019) and are given by $p_0 \sim \mathcal{U}(10^{-4}, 0.5)$, $\sigma_p \sim \mathcal{U}(10^{-4}, 0.3)$, $\rho_p \sim \mathcal{U}(10^{-4}, 0.99)$, $\phi_p^+ \sim \mathcal{U}(10^{-4}, 0.5)$, $\phi_p^- \sim \mathcal{U}(10^{-4}, 0.5)$, $n_0 \sim \mathcal{U}(10^{-4}, 1)$, $\sigma_n \sim \mathcal{U}(10^{-4}, 0.3)$, $\rho_n \sim \mathcal{U}(10^{-4}, 0.99)$, $\phi_n^+ \sim \mathcal{U}(-0.2, 0.1)$, $\phi_n^- \sim \mathcal{U}(10^{-4}, 0.75)$, and $\mu_t \sim \mathcal{U}(-0.9, 0.9)$.

The major challenge in this inference problem is the computational cost associated with the evaluation of the log-likelihood function,

$$
\log \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \log \mathcal{L}(\boldsymbol{\theta}; R_{\text{obs},T}) = \sum_{t=1}^{T} \log \pi(r_{\text{obs},t} \mid r_{\text{obs},t-1}, \boldsymbol{\theta}). \tag{4.9}
$$

Bekaert et al. (2015) show that evaluation of the transitional densities, $\pi(r_{\text{obs},t} \mid r_{\text{obs},t-1}, \boldsymbol{\theta})$, requires the so called BEGE density $\pi_{\text{BEGE}}(u_t \mid \sigma_p, \sigma_n, p_{t-1}, n_{t-1})$. This can be approximated by computing the BEGE distribution function and taking finite differences (Bekaert et al., 2015). The BEGE distribution function is given by

$$
F_{\text{BEGE}}(u_t \mid \sigma_p, \sigma_n, p_{t-1}, n_{t-1}) = \int_{-\infty}^{\infty} G_{\tilde{\Gamma}}(\omega_{p,t} - u_t \mid n_{t-1}, \sigma_n) \pi_{\tilde{\Gamma}}(\omega_{p,t} \mid p_{t-1}, \sigma_p) \, d\omega_{p,t}, \tag{4.10}
$$

with $G_{\tilde{\Gamma}}(\omega_{p,t} - u_t \mid n_{t-1}, \sigma_n) = 1 - F_{\tilde{\Gamma}}(\omega_{p,t} - u_t \mid n_{t-1}, \sigma_n)$, where $\pi_{\tilde{\Gamma}}(\cdot \mid k, s)$ and $F_{\tilde{\Gamma}}(\cdot \mid k, s)$ are, respectively, the probability density and distribution functions of a centered gamma distribution with shape k and scale s. The integral in Equation (4.10) can be approximated numerically using quadrature. For each point in the discretisation of the $\omega_{p,t}$ parameter space, we require two evaluations of the incomplete gamma function,

$$
P(a, x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} \, dt, \tag{4.11}
$$

where $\Gamma(a)$ is the gamma function. Equation (4.11) can be computed using a power series (Press et al., 1992).

We apply an adaptive SMC sampler to move N_p particles from the prior $\pi(\boldsymbol{\theta})$ to the posterior $\pi(\boldsymbol{\theta} \mid R_{\mathrm{obs},T})$ under the BEGE model. This is a very similar SMC sampler to that applied in Section 4.1. The main difference is the scaling rule for the proposal kernel within the MCMC step, as the posterior is highly non-Gaussian. We apply the method of Salomone et al. (2018) to evaluate a set of MCMC trials each with a random scale factor, $h \in [0.1, 0.2, \ldots, 1.0]$, at each SMC iteration. We then choose the scale factor, h_{opt}, which maximises the median expected squared jump distance across all particles. This continues until at least half of the particles have moved further than the median (Salomone et al., 2018).

Parallelisation and vectorisation opportunities

In the BEGE model inference problem, we utilise both SIMD and multithreading to accelerate a single SMC sampler. Parallel implementations of SMC samplers and particle filters have been well studied in the literature (Hurn et al., 2016; Lee et al., 2010a; Murray et al., 2016). Within a single SMC iteration, particles are completely independent of each other. However, as noted in Section 4.1, automatic synchronisation can occur within the MCMC proposal mechanism. Rather than distribute N_p particles across P cores, we ensure that the distribute occurs in contiguous blocks of length V. Each core will process $N_p/(PV)$ blocks of particles. All data associated with particles is processed in contiguous blocks of length V, allowing each thread to independently exploit SIMD within the MCMC proposal mechanism as per Algorithm 3.

Another way to exploit SIMD is in the evaluation of the BEGE log-likelihood. We extend the approximation of the integral in Equation (4.10). Consider a discretisation of $\omega_{p,t}$ of $N_\omega + 1$ nodes with spacing $\Delta\omega$. Then Equation (4.10) can be approximated by

$$
\begin{aligned}
F_{\mathrm{BEGE}}(u_t \mid \sigma_p, \sigma_n, p_{t-1}, n_{t-1}) \approx \sum_{j=1}^{N_\omega} &\left\{ \left[1 - P\left(n_{t-1}, \frac{\omega_{p,t}^{j-1} - u_t + n_{t-1}\sigma_n}{\sigma_n} \right) \right] \right. \\
&\left. \times \left[P\left(p_{t-1}, \frac{\omega_{p,t}^{j} + p_{t-1}\sigma_p}{\sigma_p} \right) - P\left(p_{t-1}, \frac{\omega_{p,t}^{j-1} + p_{t-1}\sigma_p}{\sigma_p} \right) \right] \right\},
\end{aligned}
\tag{4.12}
$$

where $\omega_{p,t}^{j} = \omega_{p,t}^{0} + j\Delta\omega$, for $j = 0, 1, \ldots, N_\omega$ and $\omega_{p,t}^{0}$ is the lower bound of the discretisation. For every point $\{\omega_{p,t}^{j}\}_{0 \leq j \leq N_\omega}$, the incomplete gamma function is evaluated twice; once with p_{t-1} and once with n_{t-1}. Therefore, we can consider SIMD for the incomplete gamma function for blocks of $\omega_{p,t}$ points of length V, that is, $P^V(a, x_{1:V}) = [P(a, x_1), P(a, x_2), \ldots, P(a, x_V)]$. We achieve this by extending the method of Press et al. (1992), resulting in the element-wise vector series expression

$$
\begin{aligned}
P^V(a, x_{1:V}) = &\left[\left(\exp^V(-x_{1:V}) \circ (x_{1:V})^{\circ a} \right) \oslash (\Gamma(a)\mathbf{e}_{1:V}) \right] \\
&\circ \left[\sum_{j=0}^{\infty} \left((x_{1:V})^{\circ j} \right) \oslash ((j+1)\mathbf{e}_{1:V}) \right],
\end{aligned}
\tag{4.13}
$$

where $\exp^V(-x_{1:V}) = [e^{-x_1}, e^{-x_2}, \ldots, e^{-x_V}] \in \mathbb{R}^{1 \times V}$, and $\mathbf{e}_{1:V} = [1, 1, 1, \ldots, 1] \in \mathbb{R}^{1 \times V}$. Equation 4.13 allows efficient iteration that reuses previous steps, thus enabling good use of L1 and L2 cache. The series is truncated once it has converged under the ∞-norm.

We proceed to approximate $F_{\mathrm{BEGE}}(u_t \mid \sigma_p, \sigma_n, p_{t-1}, n_{t-1})$ by: 1) applying Equation (4.13) across the discretisation, $\{\omega_{p,t}^j\}_{0 \leq j \leq N_\omega}$, in blocks of length V with shape p_{t-1}; 2) applying Equation (4.13) across the discretisation, $\{\omega_{p,t}^j\}_{0 \leq j \leq N_\omega}$, in blocks of length V with shape n_{t-1}; and 3) accumulating the sum of products in Equation 4.12.

Performance

We implement the BEGE inference problem using adaptive SMC with likelihood annealing with $N_p = 1024$ particles. We approximate $F_{\mathrm{BEGE}}(u_t \mid \sigma_p, \sigma_n, p_{t-1}, n_{t-1})$ using Equations (4.12) and (4.13) with the discretisation $\{\omega_{p,t}^j\}_{0 \leq j \leq N_\omega}$, $N_\omega = 100$, $\omega_{p,t}^0 = 10^{-4} - p_{t-1}\sigma_p$, and $\Delta\omega = (10\sigma_p\sqrt{p_{t-1}} - \omega_{p,t}^0)/(N_\omega - 1)$, as is performed by Bekaert et al. (2015) and South et al. (2019). Results are provided in Figure 5.

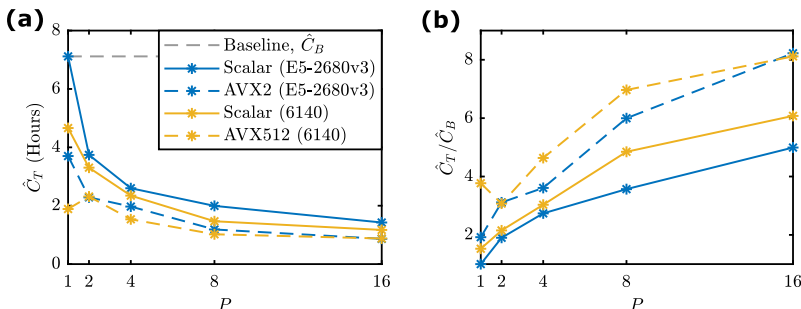
Figure 5: Benchmarking results for the SMC sampler for BEGE parameter inference. (a) Mean runtimes, $\hat{C}_T$, using P cores for various implementations: C with scalar operations only (Xeon E5-2680v3; blue solid), C with AVX2 vectors (Xeon E5-2680v3; blue dashed), C with scalar operations only (Xeon Gold 6140; yellow solid), and C with AVX512 vectors (Xeon Gold 6140; yellow dashed). (b) Computational improvements for each optimisation relative to baseline $\hat{C}_B$ (C with scalar operations only with $P = 1$ on Xeon E5-2680v3).

We observe an improvement of up to $2\times$ for AVX2 and almost $4\times$ for AVX512 regardless of the number of cores. This is improved VPU utilisation for AVX512 compared with the weak informativity test (Section 4.1). That is, the BEGE distribution function approximation is a sufficiently large proportion of the total computation cost, that efficient calculation can exploit the longer 512 bit vectors. However, the overall speedup factor is lower from multithreading and increases slowly with P. The SMC sampler requires threads to synchronise at the end of each iteration. This synchronisation is in the resampling and annealing steps, and in estimating the optimal MCMC proposal

scaling could be causing bottlenecks. This can be seen in the diminishing returns on the parallel speed-up as the number cores increases.

Summary

We have demonstrated how SIMD can be used to further accelerate a parallel SMC sampler for a challenging inference problem from econometrics. Note that the log-likelihood approximation we apply, based on the work of Bekaert et al. (2015) is biased. Recently, Li et al. (2020) proposed an unbiased likelihood estimator, for which there are SIMD opportunities also. For the purposes of this manuscript, we find the biased approximation of Bekaert et al. (2015) lends itself more direct discourse.

5   Discussion

The example applications demonstrate common features that allow several conclusions to be drawn regarding task suitability for SIMD. Firstly, it is clear that the form of stochastic model under study can have a dramatic effect on the potential performance boost. Simulation schemes such as Euler-Maruyama are ideal candidates for SIMD, however, exact stochastic simulation methods like Gillespie's method (Gillespie, 1977) are more challenging. We show this effect in the Supplementary Material (Warne et al., 2021) using the application of ABC for analysis of Tuberculosis transmission (Tanaka et al., 2006) and observe only moderate improvements from SIMD. This highlights a difference between continuous time and discrete time Markov processes, and may motivate the practitioner to consider alternative algorithms that are more suited to SIMD, for example, the τ-leap approximation (Gillespie, 2000, 2001). The success of the SIMD version of the MCMC proposal kernel in Section 4.1 also relied on a model for which SIMD implementation of the likelihood function was possible. Secondly, each application involved nested parallelisation. The utility of SIMD here is that within each parallel thread, the computational tasks may be further sub-divided through use of VPUs. Efficiency gains due to SIMD are multiplicative to those arising from the use of thread parallelisation. Finally, each application involved a mixture of tasks that can be performed completely independently and tasks that require synchronisation or communication. This is important, since independent parallel computation is ideal for multithreading and fine grain parallel operations with frequent synchronisation are well suited for SIMD. Our demonstrations are widely applicable, since these three features are common to many Bayesian applications.

Many other Monte Carlo and Bayesian applications could benefit from these approaches. One possibility is large scale particle filters, perhaps operating within a pseudo-marginal scheme (Andrieu et al., 2010), which could be implemented as a hybrid algorithm in which proposals and weight updates are performed in SIMD blocks that are distributed across multiple threads. Such a scheme would enable prefix summations, that is, parallel computation of cumulative sums, to be performed (Hillis and Steele, 1986) in the resampling step, which may improve the performance. Another possibility is within the class of ABC validation or post-processing procedures. For example, in

the recalibration post-processing technique of Rodrigues et al. (2018), each of the N samples $\boldsymbol{\theta}^*$ from the approximate posterior distribution are individually recalibrated by the construction of a further approximate (ABC) posterior for each $\boldsymbol{\theta}^*$ using all previously generated $(\boldsymbol{\theta}^*, \mathcal{D}^*)$ pairs, but based on observing the associated (simulated) dataset $\mathcal{D}^*$, and constructing all univariate marginal posterior distribution functions. This re-use of ABC algorithm and previously generated parameter values and datasets is very common in ABC (e.g. Blum et al., 2013), and makes them particularly suited for performance gains through parallelism and SIMD. Likelihood-based and frequentist applications can also benefit from techniques we present here. For example, complex optimisations for maximum likelihood parameter estimates (Hurn et al., 2016). In general, any application that has been demonstrated using GPGPUs could exploit SIMD on the CPU (Lee et al., 2010a; Holbrook et al., 2020). This is rarely considered when comparing these technologies (Holbrook et al., 2020; Lee et al., 2010b).

We have provided example C programs, using OpenMP and MKL libraries. We appreciate that, for very good reasons, higher level languages such as Matlab and R are preferred environments for many practitioners. The techniques we present can be exploited through Matlab C-MEX or Rcpp interfaces. However, Matlab and R must be configured correctly to use the required compiler options. Matrix operations performed within high level languages, such as Matlab or R, are likely already utilising SIMD via high-performance BLAS and LAPACK libraries. Unfortunately, many of the SIMD and memory optimisations cannot be directly exploited using a high level language alone, as demonstrated in Section 3. Between successive high level functions calls, it is unlikely that the caches are preserved, and as a result, our SIMD versions of the Euler-Maruyama scheme and MCMC proposal cannot be directly replicated in Matlab or R. An exception to this rule is the Julia language (Bezanson et al., 2017) using the `@simd` macro to achieve high performance.

OpenMP is also not the only way to access SIMD within C/C++. For example, OpenCL kernels may be compiled for CPUs that support SIMD units (Hurn et al., 2016; Macintosh et al., 2019). Instruction level intrinsic functions (available in R via the RcppXsimd package) allow advanced features such as efficient random variates for small vectors, but this approach is very challenging and akin to machine code.

We have demonstrated that by following a few simple guidelines to maximise the utilisation of modern CPUs, advanced Monte Carlo methods may be relatively straightforwardly accelerated by a factor of up to $6\times$ when using SIMD compared to equivalent scalar code in C. We also show speedups of more than $25\times$ when compared with optimised R code. These speedups are all for serial code and are multiplied by further speedups obtained through multithreading. These techniques will only become more relevant in the future as CPUs architectures are released with wider VPUs and statisticians develop more complex and sophisticated inferential algorithms.

Supplementary Material

Bayesian computations using SIMD operations (DOI: 10.1214/21-BA1265SUPP; .pdf). Contains: Tutorial code using R and C and example C code using OpenMP and MKL.

References

Amdahl, G. M. (1967). "Validity of the single processor approach to achieving large scale computing capabilities." In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), 483–485. New York, NY, USA: ACM. 598

Andrieu, C., Doucet, A., and Holenstein, R. (2010). "Particle Markov chain Monte Carlo methods." *Journal of the Royal Statistical Society: Series B*, 72: 269–342. MR2758115. doi: https://doi.org/10.1111/j.1467-9868.2009.00736.x. 616

Barber, S., Voss, J., and Webster, M. (2015). "The rate of convergence for approximate Bayesian computation." *Electronic Journal of Statistics*, 9: 80–195. MR3306571. doi: https://doi.org/10.1214/15-EJS988. 599, 608

Bekaert, G., Engstrom, E., and Ermolov, A. (2015). "Bad evironments, good environments: A non–Gaussian asymmetric volatility model." *Journal of Economics*, 186: 258–275. MR3321537. doi: https://doi.org/10.1016/j.jeconom.2014.06.021. 595, 612, 613, 615, 616

Beskos, A., Jasra, A., Kantas, N., and Thiery, A. (2016). "On the convergence of adaptive sequential Monte Carlo methods." *Annals of Applied Probability*, 26: 1111–1146. MR3476634. doi: https://doi.org/10.1214/15-AAP1113. 609

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). "Julia: A fresh approach to numerical computing." *SIAM Review*, 59(1): 65–98. MR3605826. doi: https://doi.org/10.1137/141000671. 595, 617

Blum, M. G. B., Nunes, M. A., Prangle, D., and Sisson, S. A. (2013). "A comparative review of dimension reduction methods in approximate Bayesian computation." *Statistical Science*, 28: 189–208. MR3112405. doi: https://doi.org/10.1214/12-sts406. 617

Bollerslev, T. (1986). "Generalized autoregressive conditional heteroskedasticity." *Journal of Econometrics*, 31(3): 307 – 327. MR0853051. doi: https://doi.org/10.1016/0304-4076(86)90063-1. 612

Bonassi, F., You, L., and West, M. (2011). "Bayesian learning from marginal data in bionetwork models." *Statistical Applications in Genetics and Molecular Biology*, 10(1): 49. MR2851291. doi: https://doi.org/10.2202/1544-6115.1684. 599, 600

Bradley, T., du Toit, J., Tong, R., Giles, M., and Woodhams, P. (2011). "Paralleization techniques for random number generators." In Green, T. and Day, R. (eds.), *GPU Computing Gems Emerald Edition*. Elsevier Inc. 598

Chopin, N. (2002). "A sequential particle filter method for static models." *Biometrika*, 89: 539–551. MR1929161. doi: https://doi.org/10.1093/biomet/89.3.539. 609

Crago, N. C., Stephenson, M., and Keckler, S. W. (2018). "Exposing memory access patterns to improve instruction and memory efficiency in GPUs." *ACM Transactions on Architecture and Code Optimization*, 15(4): 45:1–45:23. 597

Davies, R. and Brooks, R. J. (2007). "Stream correlations in multiple recursive and congruential generators." *Journal of Simulation*, 1: 131–135. 598

Del Moral, P., Doucet, A., and Jasra, A. (2006). "Sequential Monte Carlo samplers." *Journal of the Royal Statistical Society: Series B*, 68: 411–436. MR2278333. doi: https://doi.org/10.1111/j.1467-9868.2006.00553.x. 594

Evans, M. and Jang, G. H. (2011). "Weak informativity and the information in one Prior relative to another." *Statistical Science*, 26: 423–439. MR2917964. doi: https://doi.org/10.1214/11-STS357. 593, 594, 608, 609, 610

Fan, Y. and Sisson, S. A. (2018). "ABC Samplers." In Sisson, S. A., Fan, Y., and Beaumont, M. A. (eds.), *Handbook of Approximate Bayesian Computation*, 87–123. MR3889281. 599

Gelman, A. (2006). "Prior distributions for variance parameters in hierarchical models (Comment on an article by Browne and Draper)." *Bayesian Analysis*, 1: 515–534. MR2221284. doi: https://doi.org/10.1214/06-BA117A. 593, 608

Geng, T., Diken, E., Wang, T., Jozwiak, L., and Herbordt, M. (2018). "An access-pattern-aware on-chip vector memory system with automatic loading for SIMD architectures." In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 1–7. 597

Gillespie, C. and Lovelace, R. (2017). *Efficient R Programming*. O'Reilly Media, Inc, USA. 594, 601

Gillespie, D. T. (1977). "Exact stochastic simulation of coupled chemical reactions." *The Journal of Physical Chemistry*, 81: 2340–2361. 616

Gillespie, D. T. (2000). "The chemical Langevin equation." *The Journal of Chemical Physics*, 113: 297–306. 616

Gillespie, D. T. (2001). "Approximate accelerated simulation of chemically reacting systems." *The Journal of Chemical Physics*, 115: 1716–1733. 616

Glosten, L. R., Jagannathan, R., and Runkle, D. E. (1993). "On the relation between the expected value and the volatility of the nominal excess return on stocks." *The Journal of Finance*, 48(5): 1779–1801. 612

Green, P. J., Łatuszyński, K., Pereyra, M., and Robert, C. P. (2015). "Bayesian computation: a summary of the current state, and samples backwards and forwards." *Statistics and Computing*, 25: 835–862. MR3360496. doi: https://doi.org/10.1007/s11222-015-9574-5. 594

Hillis, W. D. and Steele, G. L., Jr. (1986). "Data parallel algorithms." *Communications of the ACM*, 29: 1170–1183. 616

Holbrook, A. J., Lemey, P., Baele, G., Dellicour, S., Brockmann, D., Rambaut, A., and Suchard, M. A. (2020). "Massive parallelization boosts big Bayesian multidimensional scaling." *Journal of Computational and Graphical Statistics*, 1–14. MR4235961. doi: https://doi.org/10.1080/10618600.2020.1754226. 617

Hurn, A. S., Lindsay, K. A., and Warne, D. J. (2016). "A heterogeneous computing approach to maximum likelihood parameter estimation for the Heston model of stochastic volatility." In Nelson, M., Mallet, D., Pincombe, B., and Bunder, J. (eds.), *Proceedings of the 12th Biennial Engineering Mathematics and Applications Conference, EMAC-2015*, volume 57 of *ANZIAM J.*, C364–C381. MR3580954. 594, 597, 610, 614, 617

Jasra, A., Jo, S., Nott, D., Shoemaker, C., and Tempone, R. (2019). "Multilevel Monte Carlo in approximate Bayesian computation." *Stochastic Analysis and Applications*, 0(0): 1–15. MR3946614. doi: https://doi.org/10.1080/07362994.2019.1566006. 594

Klingbeil, G., Erban, R., Giles, M., and Maini, P. K. (2011). "STOCHSIMGPU: parallel stochastic simulation for the Systems Biology Toolbox 2 for MATLAB." *Bioinformatics*, 27: 1170–1171. 594

Lee, A., Yau, C., Giles, M. B., Doucet, A., and Holmes, C. C. (2010a). "On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods." *Journal of Computational and Graphical Statistics*, 19: 769–789. 594, 598, 610, 614, 617

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010b). "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU." *SIGARCH Computer Architecture News*, 38: 451–460. 594, 597, 617

Li, D., Clements, A., and Drovandi, C. (2020). "Efficient Bayesian estimation for GARCH-type models via sequential Monte Carlo." *Econometrics and Statistics*. In Press. 616

Macintosh, H. J., Banks, J. E., and Kelson, N. A. (2019). "Implementing and evaluating an heterogeneous, scalable, tridiagonal linear system solver with OpenCL to target FPGAs, GPUs, and CPUs." *International Journal of Reconfigurable Computing*, 2019: 1–13. 617

Mahani, A. S. and Sharabiani, M. T. A. (2015). "SIMD parallel MCMC sampling with applications for big-data Bayesian analytics." *Computational Statistics and Data Analysis*, 88: 75–99. MR3332018. doi: https://doi.org/10.1016/j.csda.2015.02.010. 594

Marjoram, P., Molitor, J., Plagnol, V., and Tavare, S. (2003). "Markov chain Monte Carlo without likelihoods." *Proceedings of the National Academy of Sciences of the United States of America*, 100: 15324–15328. 594

Maruyama, G. (1955). "Continuous Markov processes and stochastic equations." *Rendiconti del Circolo Matematico di Palermo*, 4(1): 48–90. MR0071666. doi: https://doi.org/10.1007/BF02846028. 600

Mudalige, G. R., Reguly, I. Z., and Giles, M. B. (2016). "Auto-vectorizing a large-scale production unstructured-mesh CFD application." In *Proceedings of the 3rd Workshop*

on Programming Models for SIMD/Vector Processing, WPMVP '16, 5:1–5:8. New York, NY, USA: ACM.   594

Murray, L. M., Lee, A., and Jacob, P. E. (2016). "Parallel resampling in the particle filter." *Journal of Computational and Graphical Statistics*, 25: 789–805. MR3533638. doi: https://doi.org/10.1080/10618600.2015.1062015.   594, 610, 614

Nott, D., Drovandi, C. C., Mengersen, K., and Evans, M. (2018). "Approximate Bayesian computation predictive *p*-values with regression ABC." *Bayesian Analysis*, 13: 59–83. MR3737943. doi: https://doi.org/10.1214/16-BA1033.   608, 609, 610

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press. MR1201159.   613, 614

Rodrigues, G. S., Prangle, D., and Sisson, S. A. (2018). "Recalibration: A post-processing method for approximate Bayesian computation." *Computational Statistics and Data Analysis*, 126: 53–66. MR3808389. doi: https://doi.org/10.1016/j.csda.2018.04.004.   617

Ross, R. J. H., Baker, R. E., Parker, A., Ford, M. J., Mort, R. L., and Yates, C. A. (2017). "Using approximate Bayesian computation to quantify cell–cell adhesion parameters in a cell migratory process." *npj Systems Biology and Applications*, 3(1): 9.   599

Salomone, R., South, L. F., Drovandi, C. C., and Kroese, D. P. (2018). "Unbiased and Consistent Nested Sampling via Sequential Monte Carlo." *ArXiv e-prints*. ArXiv:1805.03924 [stat.CO].   614

Sisson, S. A., Fan, Y., and Beaumont, M. (2018). *Handbook of Approximate Bayesian Computation*. Chapman and Hall/CRC Press. MR3889281.   593, 594, 599

Sisson, S. A., Fan, Y., and Tanaka, M. M. (2007). "Sequential Monte Carlo without likelihoods." *Proceedings of the National Academy of Sciences of the United States of America*, 104: 1760–1765. MR2301870. doi: https://doi.org/10.1073/pnas.0607208104.   594

South, L. F., Pettitt, A. N., and Drovandi, C. C. (2019). "Sequential Monte Carlo camplers with independent Markov chain Monte Carlo proposals." *Bayesian Analysis*, 14(3): 773–796. MR3960770. doi: https://doi.org/10.1214/18-BA1129.   612, 613, 615

Tanaka, M. M., Francis, A. R., Luciani, F., and Sisson, S. A. (2006). "Using approximate Bayesian computation to estimate Tuberculosis transmission parameter from genotype data." *Genetics*, 173: 1511–1520.   599, 616

Terenin, A., Dong, S., and Draper, D. (2018). "GPU-accelerated Gibbs sampling: a case study of the Horseshoe Probit model." *Statistics and Computing*. MR3914622. doi: https://doi.org/10.1007/s11222-018-9809-3.   594

Tian, X., Saito, H., Preis, S. V., Garcia, E. N., Kozhukhov, S. S., Masten, M., Cherkasov, A. G., and Panchenko, N. (2013). "Practical SIMD vectorization techniques for Intel®

Xeon Phi coprocessors." In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, 1149–1158.   595

Trobec, R., Slivnik, B., Bulić, P., and Robič, B. (2018). *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Cham: Springer International Publishing. MR3967068. doi: https://doi.org/10.1007/978-3-319-98833-7.   594, 596

van der Pas, R., Stotzer, E., Stotzer, E., and Terboven, C. (2017). *Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD*. MIT Press.   594, 596

Vo, B. N., Drovandi, C. C., and Pettitt, A. N. (2019). "Bayesian parametric bootstrap for models with intractable likelihoods." *Bayesian Analysis*, 14: 211–234. MR3910044. doi: https://doi.org/10.1214/17-BA1071.   600

Warne, D. J., Baker, R. E., and Simpson, M. J. (2018). "Multilevel rejection sampling for approximate Bayesian computation." *Computational Statistics and Data Analysis*, 124: 71–86. MR3787614. doi: https://doi.org/10.1016/j.csda.2018.02.009. 594

Warne, D. J., Baker, R. E., and Simpson, M. J. (2019). "Simulation and inference algorithms for stochastic biochemical reaction networks: from basic concepts to state-of-the-art." *Journal of the Royal Society Interface*, 16: 20180943.   599

Warne, D. J., Sisson, S. A., and Drovandi, C. (2021). "Supplementary Material of "Vector Operations for Accelerating Expensive Bayesian Computations – A Tutorial Guide"." *Bayesian Analysis*. doi: https://doi.org/10.1214/21-BA1265SUPP. 595, 606, 608, 609, 610, 611, 616

Zierke, S. and Bakos, J. D. (2010). "FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods." *BMC Bioinformatics*, 11: 184. 594