Parallelization of Radio processing in CORSIKA 8

A. Augusto Alves Jr and Nikos Karastathis

Presented at CORSIKA 8 Air-Shower Simulation and Development Workshop,
Karlsruhe, 12 June 2023

CORSIKA 8

- Shower simulations involve complex and computer intensive calculations.
- Many different algorithms, with varying life-times and workloads.
- Large showers require from days to weeks to process.
- Precision is often traded in order to achieve acceptable performance.

These slides present the progress done on the ongoing effort to deploy parallelism in order to accelerate CORSIKA 8 framework.

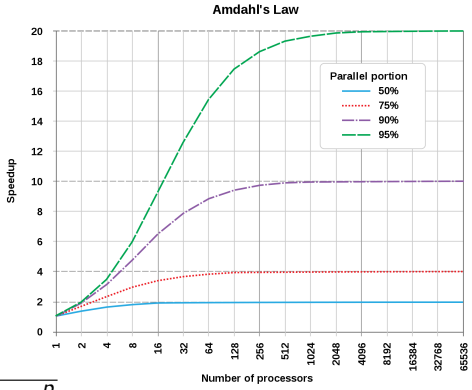
Amdahl's law

It predicts the expected speed-up from parallelism:

```
Validity of the Single Processor Approach to
Achieving Large-Scale Computing Capabilities

Amdahl, Gene M.

AFIPS Conference Proceedings (30): 483-485 (1967)
doi:10.1145/1465482.1465560
```



It is expressed as

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

where: $S(n)$ is the speed-up in function of the number of cores/threads. n is number of cores/threads and p is the fraction of code that is parallelizable.

CORSIKA, Radio and Amdahl's law

- Radio module calculates the signal corresponding to each particle/track for each antenna of the detector.
- And it often runs as one of the final operations in the sequence. Keep in mind that currently the all algorithms in the sequence run sequentially. Same applies for the processing of the particles in the stack.
- We parallelized the calculation of the signal over the detector. It means, in a per particle/track basis, the antennas response are processed in parallel.
- The expected overall speed-up (CORSIKA wise) then depends hugely of the detector size, i.e. number of antennas in the detector. I minor, but still significantly, it also depends on the calculations in the antenna itself.
- With a large detector, the importance Radio module operations grows and tends to dominate the sequence. In such, situations the speed-up is larger. This is what Amdahl's law predicts.

Gyges

`Gyges` is a new lightweight C++20 header-only library to manage thread pooling on multicore CPUs.

- With `Gyges`, thread creation and destruction costs can be paid just once in the program lifetime.
- Threads from the pool pick-up tasks as they became available. If there is no task, the threads go sleeping.
- Tasks can be submitted from multiple threads.
- The submitter gets a `std::future` for monitoring the task in-place.
- Task assignment and running can be halted at any time, from any thread.
- A `gyges::gang` can be created with any number of threads.

Status: Released and available here:

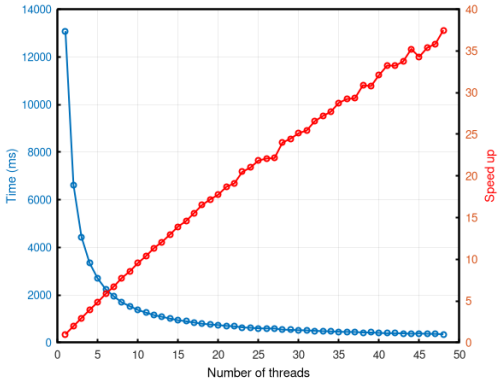
`https://gitlab.iap.kit.edu/AAAlvesJr/Gyges`

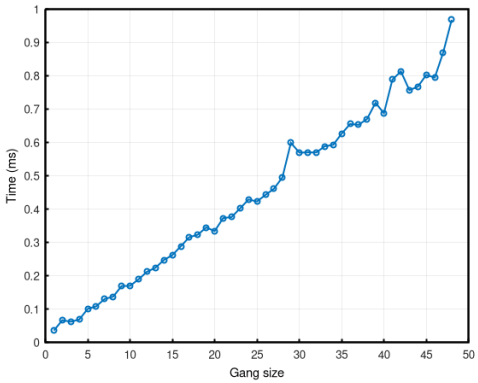
Performance and scaling

- The cost for creation and destruction of `gyges::gang` objects with different sizes has been measured using a `AMD Ryzen Threadripper 3960X 24-core/48-thread` processor running at 2.2 GHz with frequency boost enabled.
- Using same processor scalling behavior has been measured for the task below, performed 2048 times and distributed over different numbers of threads.

```
1  //number of random numbers to accumulate per task
2  unsigned nsamples = 1000000;// <- 1M of samples
3
4  size_t seed = 0x1234abcd;// <- seed
5  //functor (task)
6  auto functor = [ seed, nsamples ]( double& x)  {
7
8          std::mt19937_64 generator( seed );// <- random number generator
9          std::uniform_real_distribution<double> distribution(0.0, 1.0); // <- distribution
10
11         for(unsigned i=0; i < nsamples; ++i) x += distribution(generator); // <- generate and accumulate
12 };
```

Performance and scalling





On the left, the speed up factor and timing in terms of number of threads. On the right, the cost (timing) to create and destroy a `gyges::gang` with different number of threads.

Some implementations hints, not details

- `corsika::RadioProcess` instances own the `gyges::gang`. Size of it can be specified at construction time :

```
1 auto propagator = make_simple_radio_propagator(enviroment);
2 auto coreas     = make_radio_process_CoREAS(detector, propagator, nthreads);
3 auto    zhs     = make_radio_process_ZHS(detector, propagator, nthreads);
```

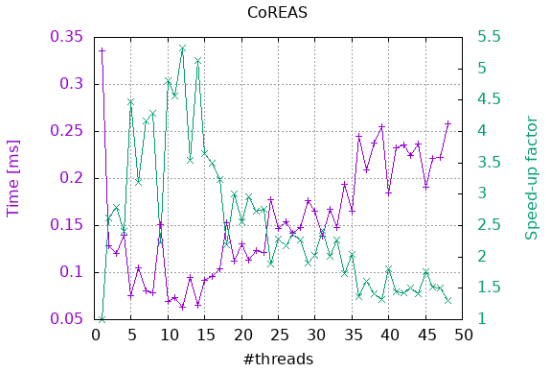
- No need to specify template parameters with the new interface.
- The implementation will define a task per thread (not per antenna). Each task will iterate over the same number of antennas, so that each thread will perform about the same amount of work.

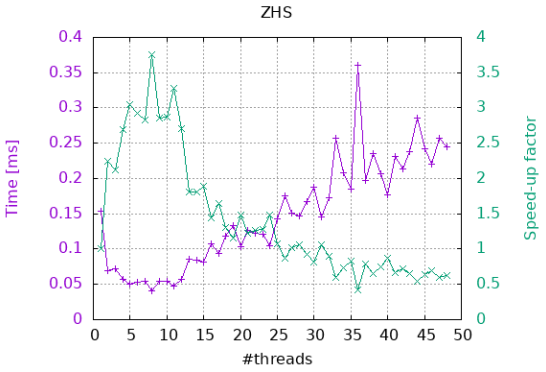
Radio process performance profile

- Profiling for scaling over the number of antennas, ie. different detector array sizes.
- Feeding just one particle associated with a single track.
- See the code snipet below

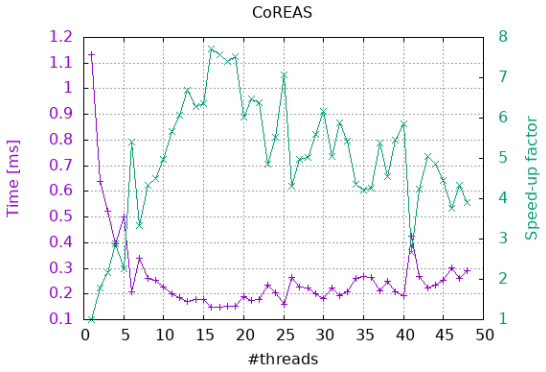
```
1  ...
2  auto propagator = make_simple_radio_propagator(enviroment);
3  auto coreas     = make_radio_process_CoREAS(detector, propagator, nthreads);
4  auto    zhs     = make_radio_process_ZHS(detector, propagator, nthreads);
5  ...
6  //start chronometer
7  coreas.doContinuous(particle, base, true);
8  //stop chronometer
9  ...
10 //start chronometer
11 zhs.doContinuous(particle, base, true);
12 //stop chronometer
```

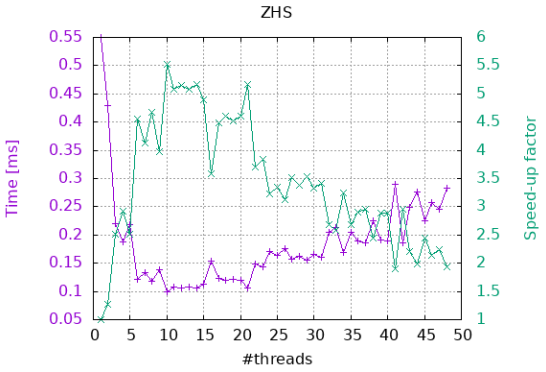
Performance: detector with 200 antennas



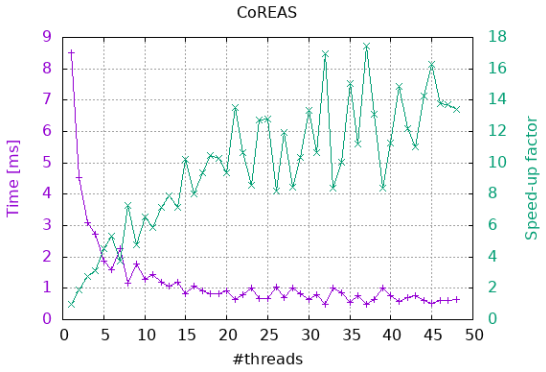


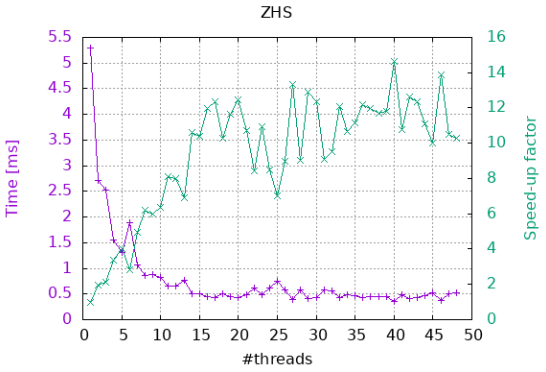
Performance: detector with 1k antennas



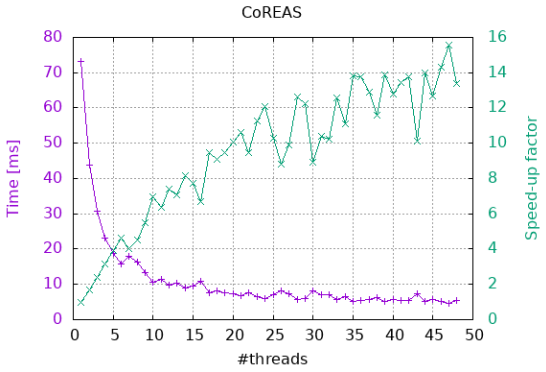


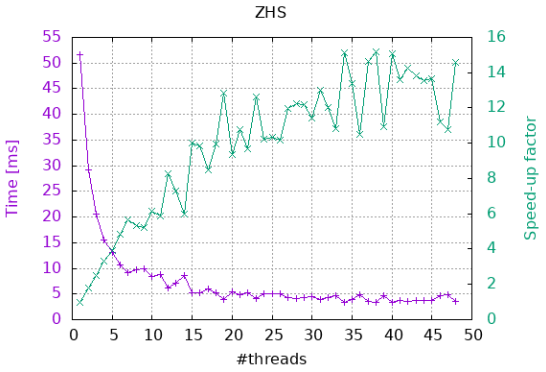
Performance: detector with 10k antennas





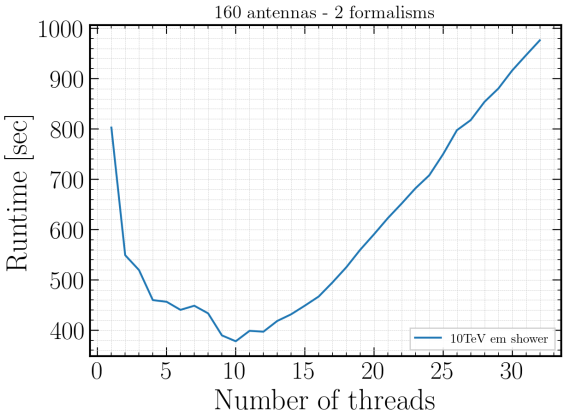
Performance: detector with 100k antennas



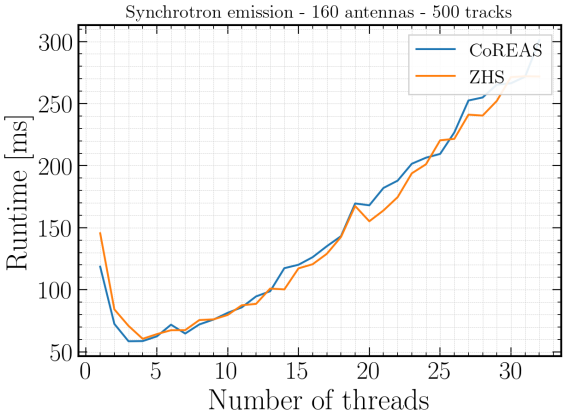


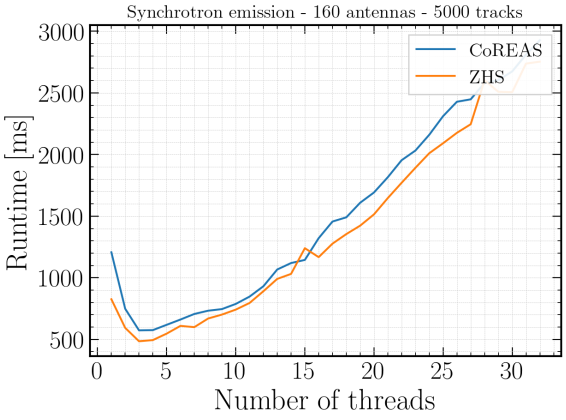
Full shower and synchrotron emission: performance profile

- On the right: full EM shower, running both algorithms with 160 antennas.
- In the next slides: profiling synchrotron emission for scaling over the number of antennas and the number of tracks
- Feeding just one particle associated in a circular trajectory.

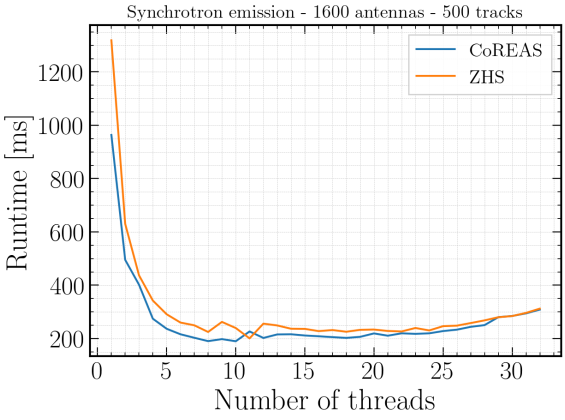


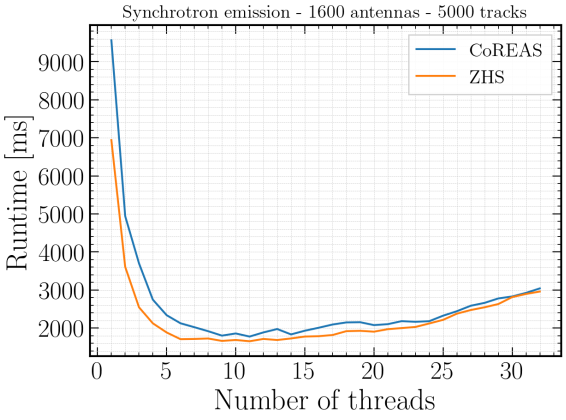
Synchrotron emission: detector with 160 antennas





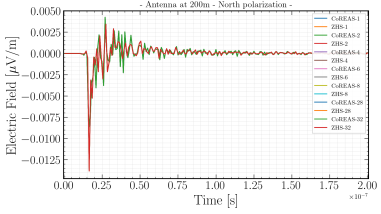
Synchrotron emission: detector with 1600 antennas

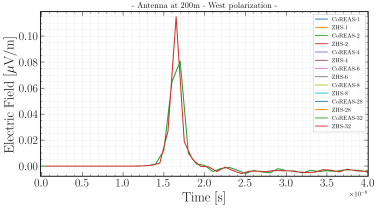


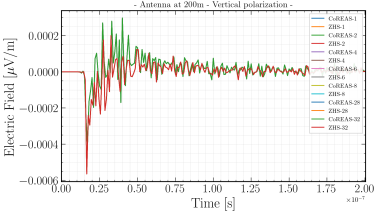


Physics output consistence

- Comparison of pulse profile between CoREAS and ZHS, for different number of threads
- Note that the lines representing the algorithms, for each number of threads, superimpose perfectly.







Conclusions

- Parallelized Radio module behave as expected, in terms of performance and physics.
- Scaling behavior for small antenna arrays should improve in case of more complexes propagators.
- One should also pay attention to the absolute value of the timing for a given number of threads, when passing from 200 to 10,000 antennas.

Backup Slides

Accelerating CORSIKA 8: a possible calculation model

Ideally, we would like CORSIKA 8 to take advantage of the availability of resources to accelerate the completion of a certain task. One possible design could be:

- To perform simulation in rounds, parallelizing the processing of the particles on each generation of the shower.
- To manage the output using side effects.
- The input data, RNG, geometry, filters etc would be services available to modules and accessible from the processing threads in read-only mode.
- The simulation manager thread would operate simultaneously an IO manager thread, a monitoring thread etc.
- The worker threads are commissioned and released by the simulation manager thread.

In summary, it is necessary an efficient, flexible and self-balancing facility to manage the task distribution and execution, in a transparent and scalable way.

Requirements

- Thread-safe modules and components.
- Fully parallel RNG.
- Thread-safe data-structure for implementing a concurrent particle stack.
- An efficient, flexible and self-balancing facility to manage the task distribution and execution, in a transparent and scalable way.

`gyges::gang` interface

```
1  class gang
2  {
3      gang(unsigned int const thread_count=std::thread::hardware_concurrency(), bool release = true ) ; // <- constructor
4      gang( gang const & other ) = delete;
5      gang( gang && other )      = delete;
6
7      template<typename FunctionType>
8      inline std::future<void> submit_task(FunctionType f) requires Dispatchable<FunctionType>; // <- submit a Dispatchable task here
9      inline void stop(void);// <- notify the running tasks (request stop), interrupt picking up new tasks and destroys the gang
10     inline std::size_t size(void);// <- get the gang size
11 };
12
13 // for_each accepting a pre-created gang
14 template<typename Iterator, typename Predicate>
15 void for_each(Iterator begin, Iterator end, Predicate const& functor,  gang& pool);
16
17 // for_each
18 template<typename Iterator, typename Predicate>
19 void for_each(Iterator begin, Iterator end, Predicate const&  functor);
```

The `concept Dispatchable` specifies signature `void operator()(std::stop_token token) noexcept`.

`gyges::concurrent_queue` interface

With this data structure, push and pop operations can be performed concurrently.

```
1  template<typename T>
2  class concurrent_queue
3  {
4      concurrent_queue();// <- constructor
5
6      concurrent_queue(const concurrent_queue& other)=delete;
7      concurrent_queue& operator=(const concurrent_queue& other)=delete;
8
9      std::shared_ptr<T> try_pop();// <- non-blocking pop
10     bool               try_pop(T& value);// <- non-blocking pop
11
12     std::shared_ptr<T> wait_and_pop(std::stop_token stoken);// <-  blocking pop
13     void               wait_and_pop(std::stop_token stoken, T& value);// <- blocking pop
14
15     void push(T value);// <- non-blocking push
16     bool empty();// <- check if empty
17 };
```

`gyges::gang` : submitting a task

```
1  ...
2  gyges::gang pool{/* hardware number of threads (2 x number of cores)*/};
3  ...
4  auto Task = [ ] (std::stop_token token) noexcept {
5
6      if(token.stop_requested() ) return; //if stop is already requested, return.
7
8      while( condition == true && token.stop_requested() == false) {
9          //do something useful in the loop
10         //ex.: generate some particles...
11         ...
12     }
13
14     //conclude and close the task
15 };
16
17 //task submission
18 auto future = pool.submit_task( Task );
19 //monitor the future
20 ...
```

Conventional pseudorandom number generators

- Most of the conventional pseudorandom number generators (PRNGs) scale poorly on massively parallel platforms (modern CPUs and GPUs).
- Inherently sequential algorithms:

$$s_{i+1} = f(s_i),$$

  where s_i is the i-th PRNG state.
- The statistical properties of the generated numbers are dependent on the function f and of the size of s_i in bits. Usually f needs to be complicated and s_i large.
- PRNGs can be deployed in parallel workloads following two approaches: *multistream* and *substream*.
- Both approaches are problematic due pressure on memory, impossibility to jump into far away states skipping the intermediate ones, correlations between streams.

Counter-based pseudorandom number generators

The so called "counter-based pseudorandom number generator" (CBPRNG) produces sequences of pseudorandom numbers following the equation

$$x_n = g(n),$$

where g is a bijection and n a counter. Basic features:

- High quality output.
- Very efficient. Actually, the designs under consideration allows to trade-off performance for efficiency in a transparent way.
- Have null or low pressure on memory, and registers, since they can be implemented in a stateless fashion.
- Very suitable for parallelism, since they allow to jump directly to an arbitrary sequence member in constant time.

Categories of CBPRNGs

- Cipher-based generators:
  - **ARS (Advanced Randomization System)** is based on the AES cryptographic block cipher and relies on AES-NI.
  - **Threefry** is based on Threefish a cryptographic block cipher and relies only on common bitwise operators and integer addition.
- Non-cryptographic bijective transformation generators:
  - **Philox**. Deploys a non-cryptographic bijection based on multiplication instructions computing the high and low halves of operands to produce wider words.
  - **Squares**. This algorithm is derived using ideas from "Middle Squares" algorithm, originally discussed by Von Neuman, coupled with Weyl sequences. Three or four rounds of squaring are enough to achieve high statistical quality. Squares implementation here is original and supports 128 bit counters with 64 bit output.

The current implementation uses ARS, Threefry and Philox from Random123 library. Squares implementation is native.

Iterator-based design for parallelism

Iterators are a generalization of pointers and constitutes the basic interface connecting all STL containers with algorithms.

- Iterators are lightweight objects that can be copied with insignificant computing costs.
- Iterator-based designs very convenient for parallelism.
- A very popular choice for implementing designs based on lazy evaluation.

These features considered all together make an iterator-pair idiom the natural design choice for handling the counters and the CBPRNG output, in combination with lazy-evaluation to avoid pressure on memory and unnecessary calculations. The current implementation uses iterators from TBB library.

Iterator-based API

The streams are represented by `Stream<Distribution, Engine>` class:

- It is thread-safe and handles *multistream* and *substream* parallelism.
- Produces pseudorandom numbers distributed according with `Distribution` template parameter.
- Handles 2^{32} streams with length 2^{64}, corresponding to 2048 PB of data, in `uint64_t` output mode.
- Compatible with C++ standard distributions.

```
1  template<typename Distribution, typename Engine>
2  class Stream
3  {
4   public:
5
6     //constructor
7     Stream( Distribution const& dist, uint64_t seed, uint32_t stream );
8
9     //stl-like iterators
10    iterator_type begin() const;
11    iterator_type end() const;
12
13    //access operators
14    return_type operator[](size_t n) const;
15    return_type operator()(void);
16    return_type operator()(size_t);
17 };
```