Parallelization of Radio module using Gyges

A. Augusto Alves Jr, with Nikos Karastathis

Presented at CORSIKA development meeting - KIT, Karlshuhe
October 13, 2022

Recap: Amdahl's law

- Predicts the expected speedup 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 speedup 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 lightweight C++20 header-only library to manage thread pooling.

- 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 just 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 stopped at any time acting over an `std::stop_token`.
- A `gyges::gang` can also be created or put in a "hold-on" state. The processing of the tasks will be postponed until it is put on "unhold" status. The threads are not keep busy-waiting, they are put to sleep until "unhold" command is sent.
- Two implementations of `gyges::for_each`. One of than able to recycle an already existing `gyges::gang`.

Status: Released. Code is available here: `https://gitlab.iap.kit.edu/AAAlvesJr/Gyges`

Some implementations hints, not details

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

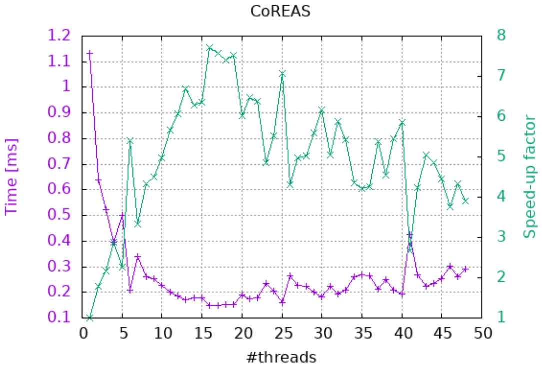
```
auto propagator = make_simple_radio_propagator(enviroment);
auto coreas     = make_radio_process_CoREAS(detector, propagator, nthreads);
auto    zhs     = make_radio_process_ZHS(detector, propagator, nthreads);
```

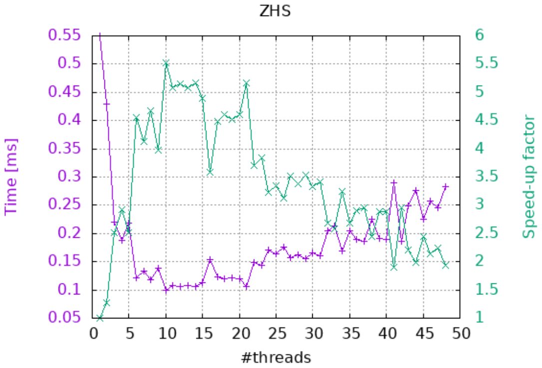
- No need to specify template parameters with the new interface.
- The implementation will define a task per thread (not per antenna). Each task will iterate over the same number of antennas, so that each thread will perform about the same amount of work.

What is being profiled?

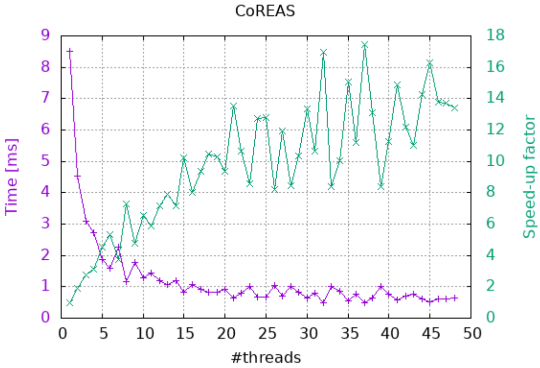
```
...
auto propagator = make_simple_radio_propagator(enviroment);
auto coreas     = make_radio_process_CoREAS(detector, propagator, nthreads);
auto    zhs     = make_radio_process_ZHS(detector, propagator, nthreads);
...
//start chronometer
coreas.doContinuous(particle, base, true);
//stop chronometer
...
//start chronometer
zhs.doContinuous(particle, base, true);
//stop chronometer
```

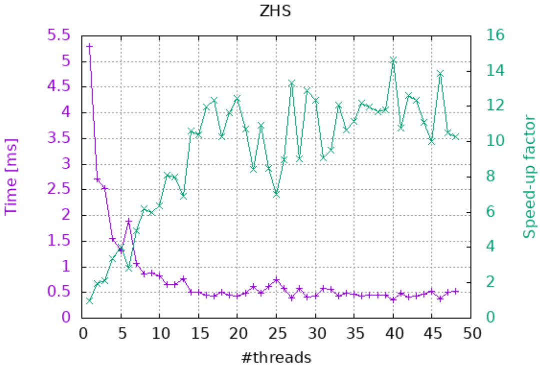
Performance: detector with 1k antennas



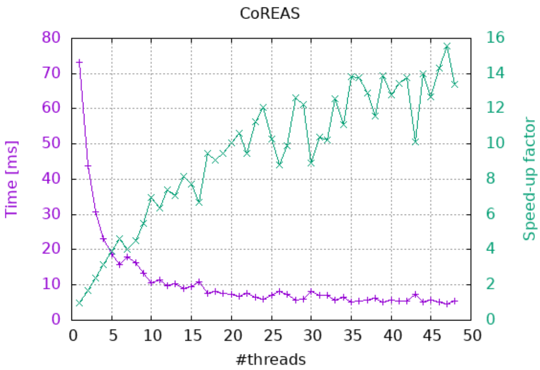


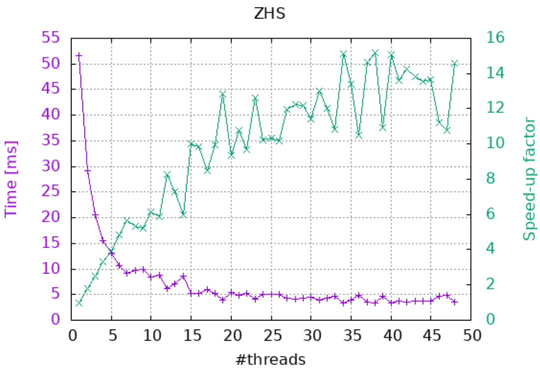
Performance: detector with 10k antennas



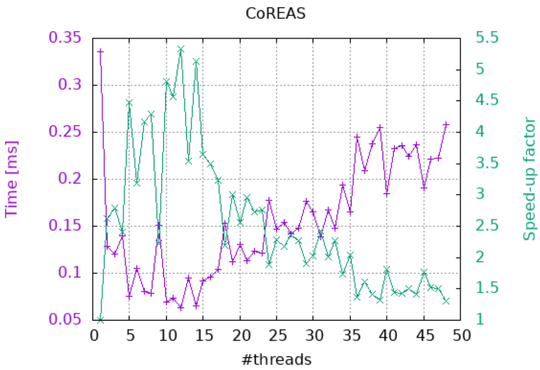


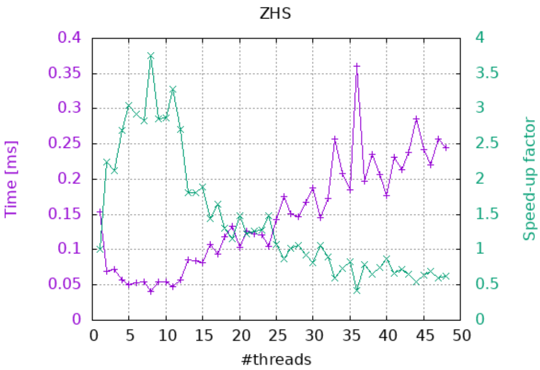
Performance: detector with 100k antennas





Performance: detector with 200 antennas





Comments

- Parallelized Radio module behave as expected, in terms of performance and physics (See Nikos' slides).
- Scaling behavior for small detectors probably improves 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.

A performance issue found during this work

- In initial stages of this study we found out the opposite to the expected behavior: time always increasing with the number of threads.
- Investigating further we found out that the culprit was the `std::shared_ptr<>` managing the the coordinate system.
- This pointer is shared among all geometry objects. By distributing this pointer among different threads we was distributing a lock.
- We substituted it by an `corsika::dumb_ptr` and the glitch went away and we even got some performance gain in single thread application.

Backup Slides

Gyges example

```
#include <future>
#include <iostream>
#include <random>
#include <vector>
#include <gyges/gang.hpp>


int main(int argv, char** argc)
{
        //number of random numbers to accumulate per task
        unsigned max_nr = 1000000000;

        // it will create a gang with the number
        // of cores supported by the hardware.
        gyges::gang thread_pool{};

        std::cout << "The gang has #" << thread_pool.size() << " workers\n";

        //tasks will accumulate max_nr of random numbers
        //and set the result in the corresponding position of a vector

        std::vector<double> results(thread_pool.size(), 0.0);
        std::vector<std::future<void>> monitors;
```

Gyges example

```
for(std::size_t i=0; i< thread_pool.size() ; ++i)
{
    //used to obtain a seed for the random number engine
    std::random_device rd;
    auto seed = rd();
    //where to place the result
    auto result_iterator = results.begin() + i;

    //lambda function getting the necessary parameters to perform the task.
    auto Task = [ result_iterator, max_nr, seed ](std::stop_token t) {

        double partial_result = 0;
        std::mt19937 generator( seed );
        std::uniform_real_distribution<double> distribution(0.0, 1.0);

        for( unsigned nr = 0; nr< max_nr; ++nr)
        partial_result+=distribution(generator);
        //set results
        *(result_iterator) = partial_result;
    };
    // task submission
    auto future = thread_pool.submit_task( Task );
    monitors.push_back( std::move(future) );

}//close for loop
```

Gyges example

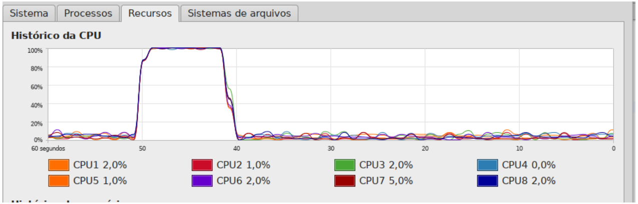
```
        //check the tasks and  print the result
        for(std::size_t i=0; i< monitors.size(); ++i  ){
                monitors[i].get();
                std::cout << "Task #" << i  << " completed. Result: "<<        results[i] << std::endl;
        }

    //stop the gang or let it get destroyed exiting scope
        thread_pool.stop();

        return 0;
}
```

Gyges example

```
[augalves@LabHome Gyges_Proj] $ ./examples/use_gangs
The gang has #8 workers
Task #0 completed. Result: 4.99999e+08
Task #1 completed. Result: 4.99998e+08
Task #2 completed. Result: 4.99998e+08
Task #3 completed. Result: 4.99975e+08
Task #4 completed. Result: 4.99992e+08
Task #5 completed. Result: 5.00011e+08
Task #6 completed. Result: 5.00009e+08
Task #7 completed. Result: 4.99997e+08
[augalves@LabHome Gyges_Proj] $
```



Basically 6x10^9 calls to RNG plus the accumulation operation performed in about 10s.

What next ?

Profiling...

Thanks