Gyges: a C++20 library for multithreading

A. Augusto Alves Jr

Presented at 5th CORSIKA 8 Parallelism & Performance Meeting - KIT, Karlshuhe

January 27, 2022

Introduction: 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.

Introduction: Comments

- Processing particles in parallel and scaling over the number of cores/threads should follow closely the Amdahl's law even if the number of particles to process is bigger than the number of cores/threads
- To achieve this behavior, dynamical job submission/monitoring and smart thread pooling should be deployed together.
- Each simulation round would be processed in parallel. The processing time will be dominated by the longer lasting job.
- Given jobs have durations spanning over a range, at a given round some threads can process more jobs, while others are busy with longer tasks.
- Currently, each particle is processed sequentially and the overall duration is the accumulation of each particle processing time.

Calculation model

- The simulation is managed in rounds. Simulation starts at first round, with the interaction of primary with media. The generated particles will be processed in the second round. The products of this round will be processed at third round... and so on.
- Simulation ends when a round produces no particles to be processed.
- Output is managed using side effects.
- Input data, RNG, geometry, filters etc are services available to modules, and accessible from the processing threads in read-only mode.
- The simulation manager thread can operates aside a IO manager thread, a monitoring thread etc. The worker threads are commissioned and released by the simulation manager thread.

Open question: How to ensure repeatability of the results calling the RNG concurrently (out of the order) ?

Gyges

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

- With `Gyges`, thread creation and destruction costs are 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 sleep.
- 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 interrupted at any time.
- A `gyges::gang` can be created with any number of threads.

Status: In final development stage. Already usable and available here:

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

Gyges example

```cpp
#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

```cpp
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

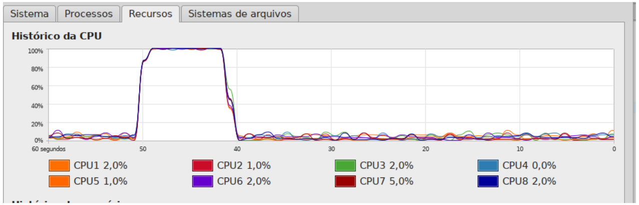
```
        //check the tasks and  print the result
        for(std::size_t i=0; i< monitors.size(); ++i  ){
                monitors[i].get();
                std::cout << "Task #" << i  << " completed. Result: "<<        results[i] << std::endl;
        }

    //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