Thoughts on parallelization in CORSIKA 8

Anatoli Fedynitch
@ ICRR

CORSIKA-8 call
January 28th 2020

Why parallelization?

1. Is **inevitable** for air-shower simulations. Running jobs on a cluster is parallelization, too.

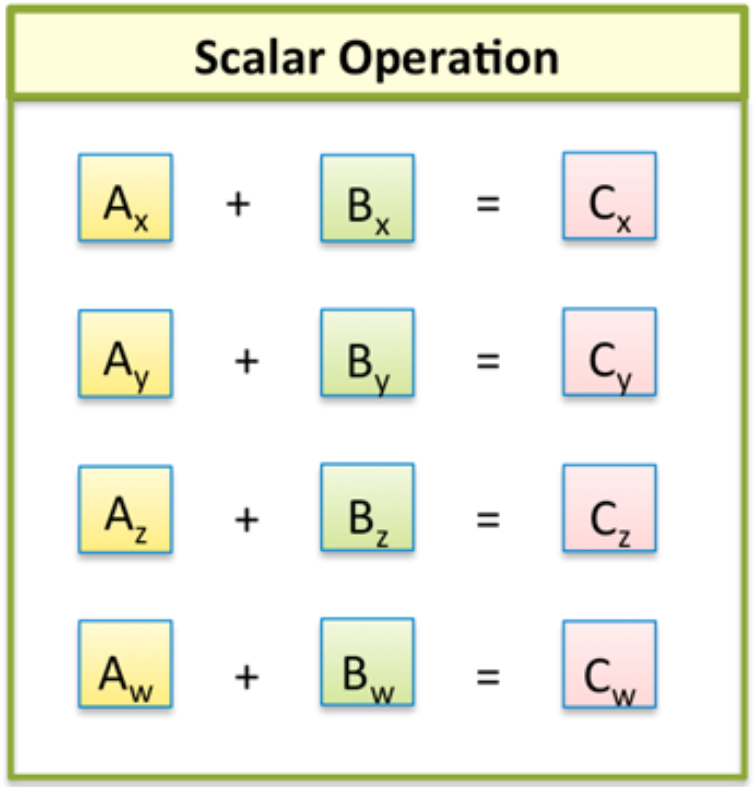
2. If it scales well to ~10k cores and good storage management, one can use supercomputers and do unthinned simulations

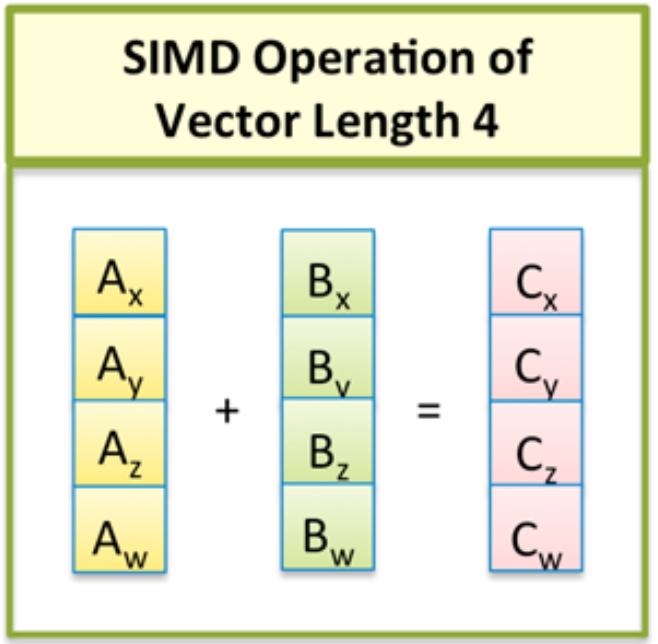
3. To use GPUs (and one should as you will see), smart **load balancing** is crucial since # GPUs << # cores per node

What are the options?

	Vectorization	Multi-threading/-processing	Naïve techniques
In simple words	Group similar data in a special way and process it one go	Multiple independent tasks, controlled by your software	Make a bunch of (single-threaded) jobs and leave it to the batch system
Who's responsible for good performance	1. **You** 2. CPU 3. OS 4. Batch size 5. past and future of your process 6. Available memory bandwidth	1. **You** 2. OS 3. CPU	1. **You** 2. Scheduler (PBS, condor, etc.) 3. Other strangers that run their tasks on same cluster 4. OS + CPU that deals with yours and the other's tasks simultaneously

On vectorization or Single Instruction Multiple Data (SIMD)





Intel® Architecture currently has SIMD operations of vector length 4, 8, 16

Refresh from ISAPP school: https://indico.cern.ch/event/719824/

Should be pretty fast, right?

```
      SUBROUTINE MATMULOPT(M, N, DATA, VEC, RES)
        INTEGER M, N, I, J
        DOUBLE PRECISION DATA(10000,10000)
        DOUBLE PRECISION VEC(10000), RES(10000)
    ' intent(out) :: RES

        DO J=1,N
            DO I=1,M
                RES(J) = DATA(I,J)*VEC(I) + RES(J)
            END DO
        END DO

      END
```

- This example is brute force
- Run on a tablet, workstation typically higher gain
- Linear algebra has many interesting features (sparse matrices, efficient solvers, etc.)

```
In [3]: m,n, data, vec = 10000,10000, np.random.random((10000,10000)), np.random.random(10000)

In [4]: dataf = np.asfortranarray(data)

In [5]: vecf = np.asfortranarray(vec)

In [6]: %timeit fortrantest.matmulopt(m,n,dataf,vecf)
10 loops, best of 3: 130 ms per loop

In [7]: %timeit np.dot(data.T, vec)
10 loops, best of 3: 35.4 ms per loop
```

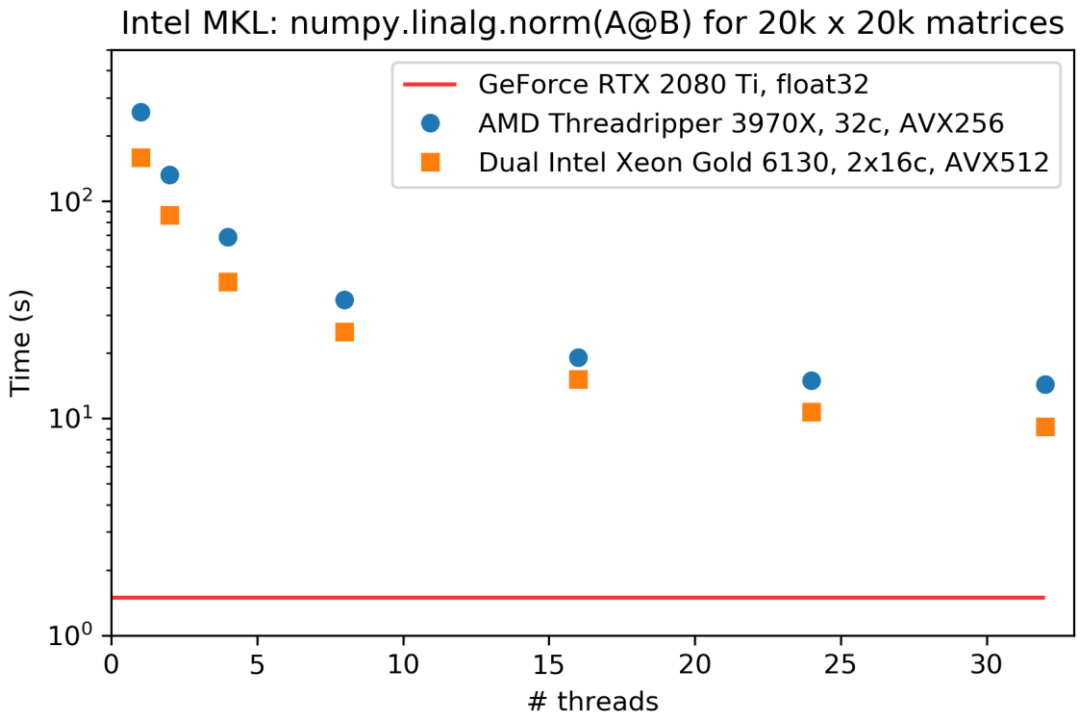
gfortran-7 -O3 vs. numpy linked to Intel MKL

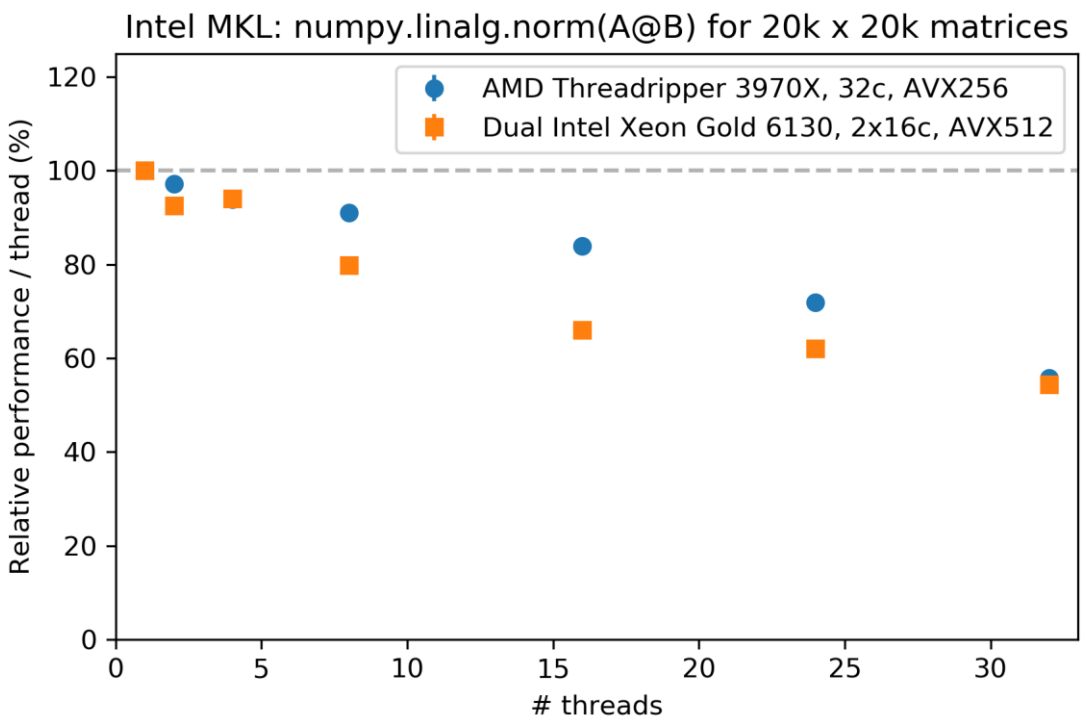
Today we'll look at two modern "HPC" machines

- **AMD Threadripper 3970X**
  - custom built
  - 32 cores, 64 threads (SMT)
  - 3.7 GHz all-core, <4.5 GHz turbo
  - Rome architecture (2019) max. 64 cores/socket
  - 1 NUMA node = "all CPUs see each others memory in the same way"
  - 128 MB L3 and 16MB L2 cache
  - Quad-channel 128 GB DDR4 3200MT/s
  - Nvidia GeForce 2080 Ti 11GB/s

- **Dual Intel Xeon Gold 6130**
  - Dell PowerEdge 1000e
  - 2 sockets with each
  - 16 cores, 32 threads (SMT)
  - 2.1 GHz with 3.9 GHz turbo
  - Total 64 threads on 2 NUMA nodes
  - 2x6-channel 192GB DDR4 2933MT/s
  - 2x22MB L3 cache
  - No GPU

Vectorization scaling with # shared-memory threads

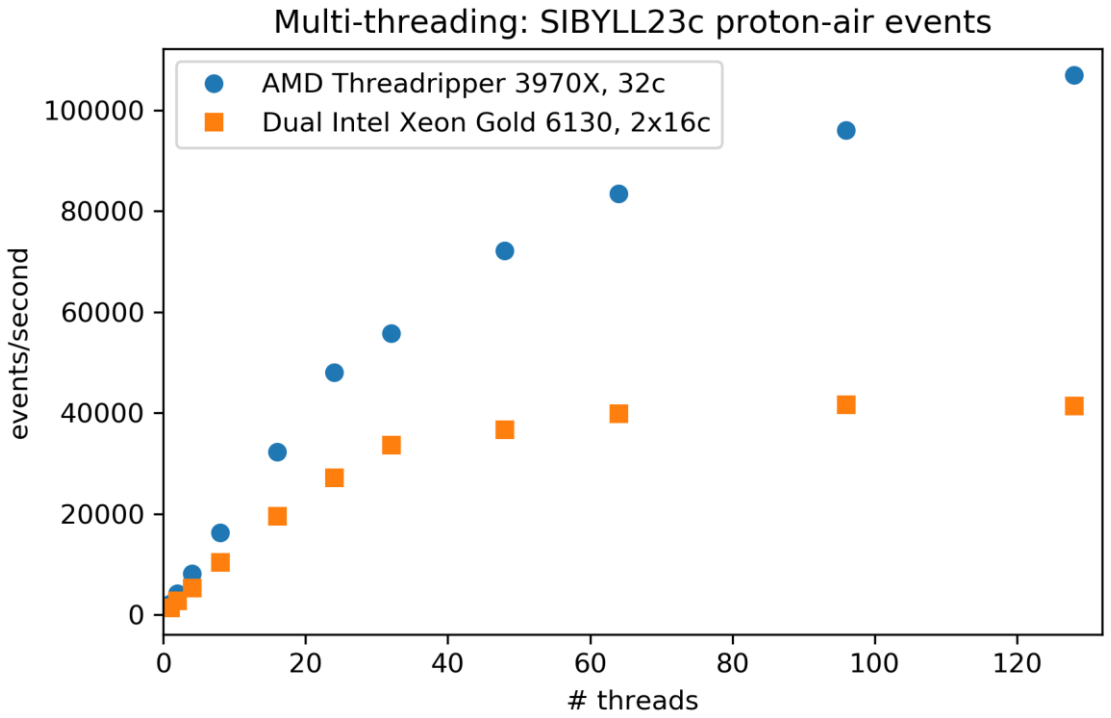


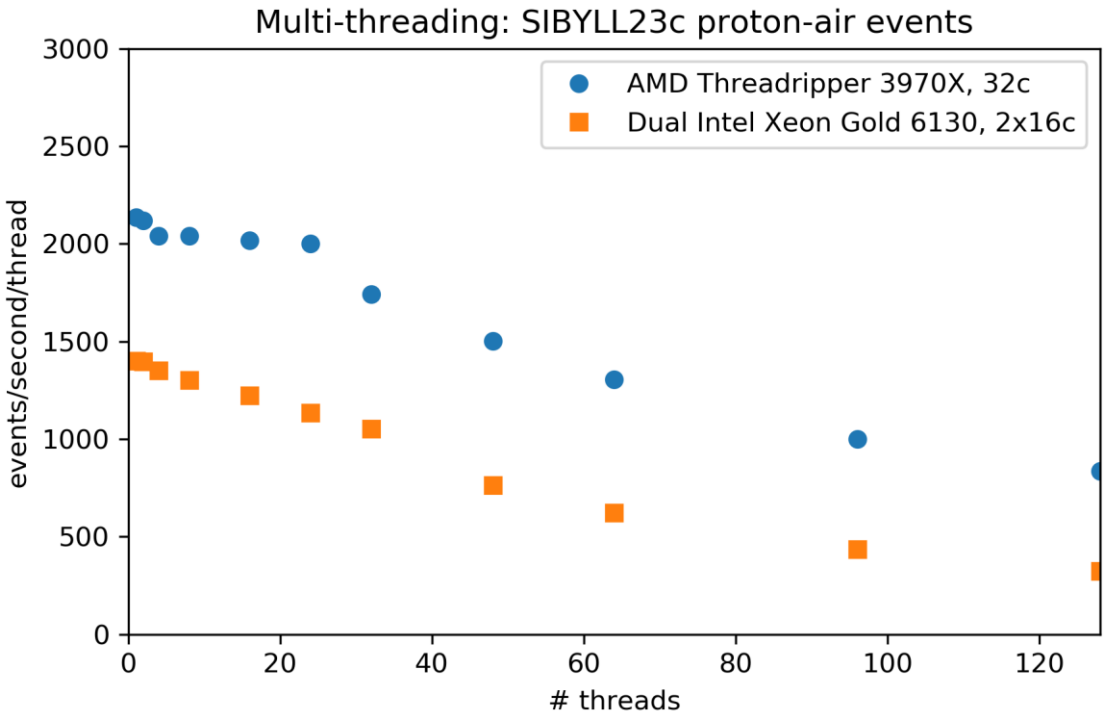


- This is **one simplistic example.** More complex examples typically scale worse. Time measures the matrix norm calculation.
- Uses 256-bit (AMD) and 512-bit SIMD units and shared memory between threads.
- Can not run as independent jobs because 1 matrix ~ 9GB

- AMD runs out of memory bandwidth at ~16 threads
  - 4-channel DDR3200 vs. “12”-channel DDR2933
- But not even close to 1 GPU @ 32-bit floats
- GPU is a specialized SIMD/vectorization architecture

Multi-processing (naïve or multi-threaded)

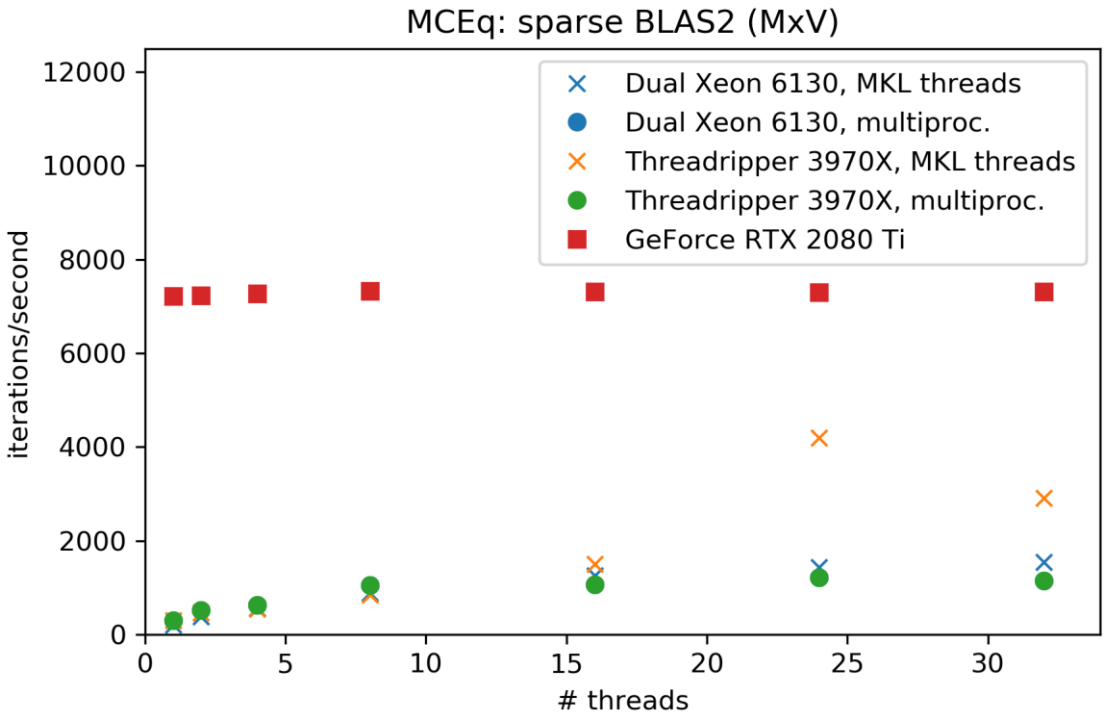


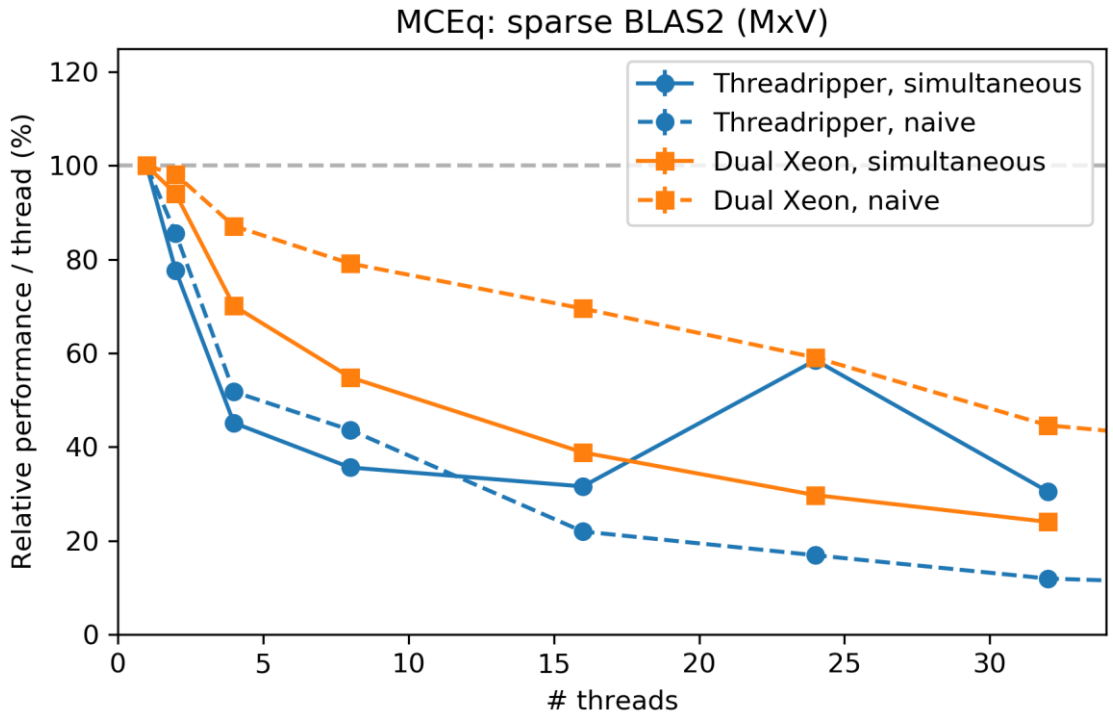


- Run up 128 SIBYLL event generators as independent processes
- For the fans: AMD drives circles around Intel ☺
- **Scales far beyond number of physical CPUs!**
- 25% lost on a batch system because typically # nodes = # threads or # cores…

- CPU logic and OS optimized extremely well because this is the most common scenario for software
- As a programmer one may save time to think about how the CPU is handling your code in global context (at the scale of a single loop or so is fine…). [No premature micro-optimization]
- This is why large scale HEP MC is run in simple naïve jobs

Multi-processing vs SIMD

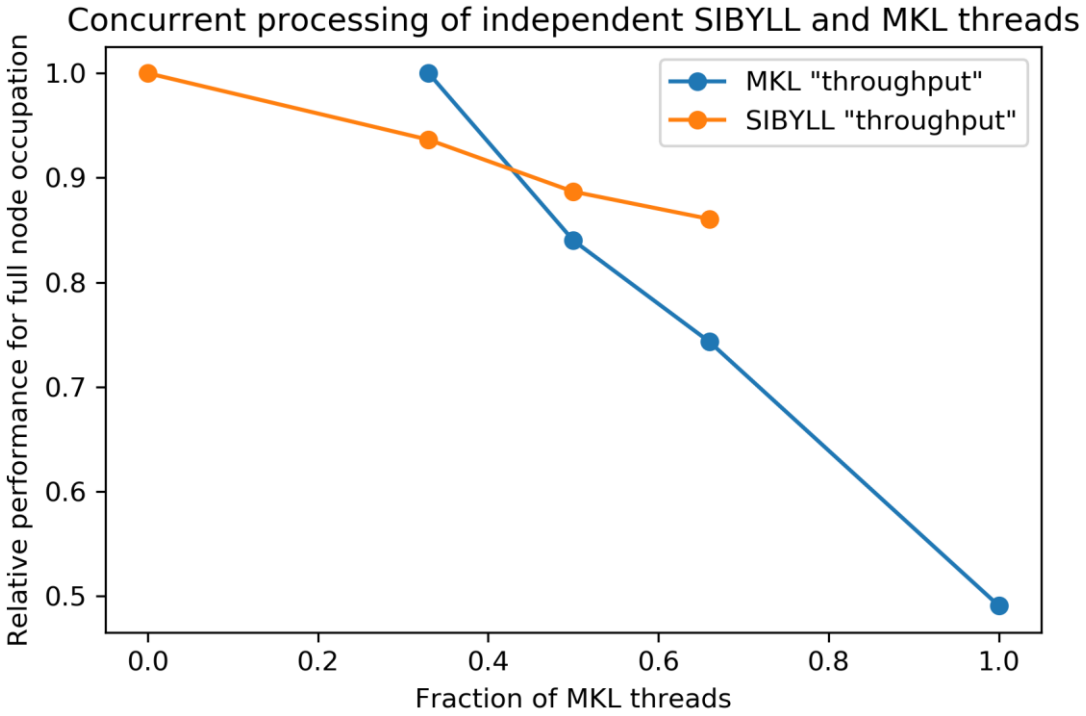


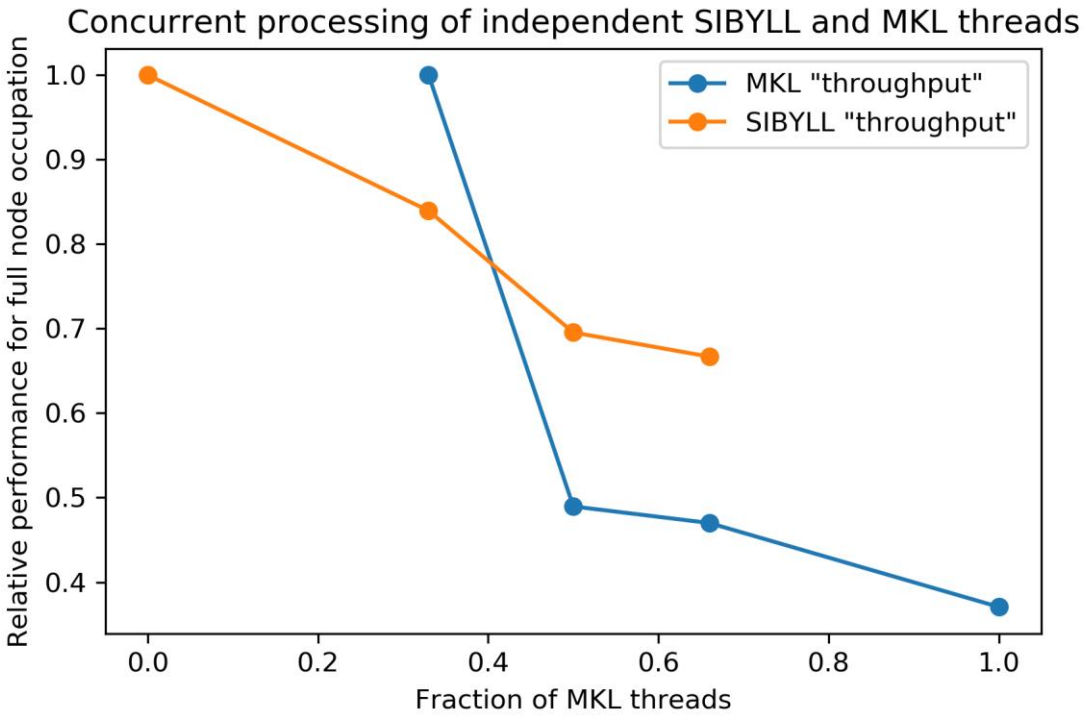


- Multiproc.: # threads of independent MCEq runs
  - Simulates “naïve” situation
- MKL threads: # threads for sp_dgemv (x = Ax + b)
  - Simulates “managed multi-processing”

- As usual: GPU drives circles around AVX CPUs in single precision
- Sparse dgemv is memory throughput limited
- Intel system wins hands down with 12 vs 4 memory channels
- Dual AMD EPYC has 2x8-channel 3200MT/s memory. Results may look very different
- Bottom line: the scaling of vectorization is architecture dependent and needs “dynamic” runtime optimization

Mixed concurrent workload (naïve)





- Run 64 SIBYLL threads on Threadripper
- Add a fraction of single-threaded MKL or MCEq simulations to the pool
- Simulates workload on a cluster where different users run different stuff on the same node

- MCEq more memory bound
- Typical simulation may run 30% slower with only a few “MCEq-like” threads on the same node
- Sparse gemv runs slower if memory not shared
- This would not affect GPU calculations at all

Conclusions

- **From the SIBYLL examples: Have faith in hardware! It's gonna handle what you throw at it better then you would expect.**
- **From vectorization: everything is difficult!**
  - There is a huge performance gain (ISAPP example)
  - But it is only in the ideal case. Typical scenario for CORSIKA is not your laptop. These are clusters with many concurrent workloads. Depends on architecture.
  - Memory bandwidth is an issue and competing processes may interfere and destroy the performance gain.
  - And all this ignores the fact that the entire simulation has to be specifically designed to group the data in SIMD-efficient way. GEANT V failed to do it.
  - GPU is the ultimate vectorized architecture. There is much more fruit, although they are more difficult to get.
- **On naïve parallelization:**
  - It's not as bad as it sounds!
  - But works efficiently only for old-school tasks, like MC event generation.
  - For mixed CPU+GPU work one needs to maintain full control and load balancing. No naïve parallelization possible.
- **Scaling up CORSIKA simulations and use modern methods means that parallelization has to be managed**
  - This means that there is "no main loop" ☹
  - Instead, there has to be a scheduler.
  - More on this: next time