Calculating the Sum of a Vector

1. Single-threaded summation

Performance of all single-threaded summations

Operating System (Compiler)	Range-based for loop	std::accumulate	Locks	Atomics
Linux (GCC)	0.07	0.07	3.34	1.34 1.33
Windows (cl.exe)	0.08	0.03	4.07	1.50 1.61

- Atomics are 12 - 50 times slower on Linux and Windows than `std::accumulate`.
- Atomics are 2 - 3 times faster on Linux and Windows than locks.
- `std::accumulate` seems to be highly optimized on Windows.

Calculating the Sum of a Vector

2. Multi-threaded summation with a shared variable

Performance of all multi-threaded summations

Operating System (Compiler)	std::lock_guard	atomic +=	fetch_add	fetch_add (relaxed)
Linux (GCC)	20.81	7.78	7.87	7.66
Windows (cl.exe)	6.22	15.73	15.78	15.01

- Using a shared atomic variable with relaxed semantics and calculating the sum with four threads' help is about 100 times slower than using a single thread with the algorithm `std::accumulate`.

Calculating the Sum of a Vector

3. Thread-local summation

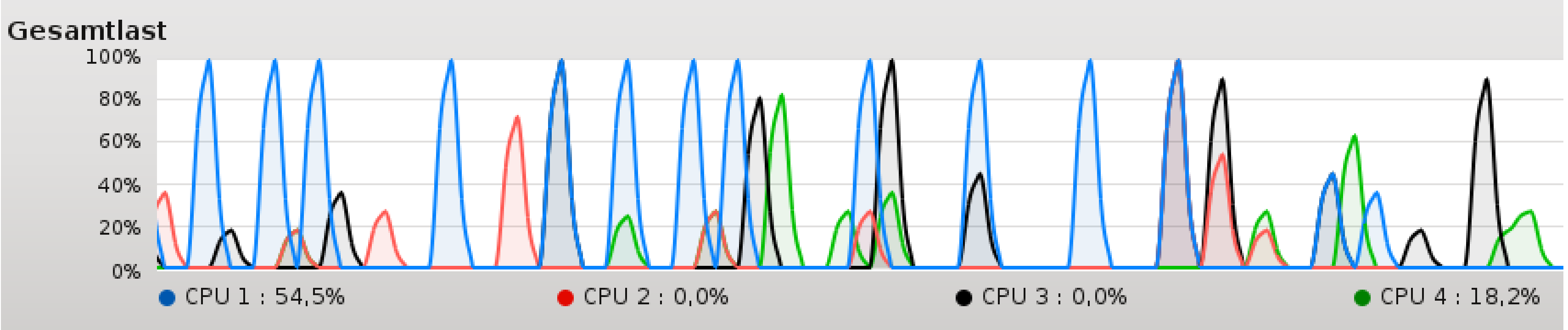
Performance of all thread-local summations

Operating System (Compiler)	std::lock_-guard	Atomic using sequential consistency	Atomic using relaxed semantics	Thread-local data	Tasks
Linux (GCC)	0.03	0.03	0.03	0.04	0.03
Windows (cl.exe)	0.10	0.10	0.10	0.20	0.10

- It makes no big difference whether I use local variables or tasks to calculate the partial sum or if I use various synchronization primitives such as atomics.
- Thread-local data seems to make the program slower.

Calculating the Sum of a Vector

1. Single threaded summation
   - The performance of range-based for loop and `std::accumulate` are similar.
2. Multithreaded summation with a shared variable
   - Synchronization is costly. Minimizing expensive synchronization must be your first goal.
3. Thread-local summation
   - The thread-local summation is only two times faster than the single-threaded range-based for loop or `std::accumulate`. The four cores are idle.



➡ The cores can't get the data fast enough from memory.