Parallelizing queue simulations

Malte Schledjewski

2nd September 2014

Abstract

A simulation to gather various statistics for different queues was implemented. The process of gathering the data was then parallelized and the scaling behaviour analysed. In the process showing why benchmarks are black magic that should always be taken with a grain of salt.

1 Simulated system

The lecture on advanced communication networks included an exercise to implement a simulation of a simple queue. The simulated system consist of:

- a queue of zero, finite or infinite length
- a service consisting of a finite number of workers
- a packet generator

Workers need a certain amount of time per packet. This time as well as the time between new packets follow a negative exponential distribution.

The goal was to gather various statistics about the system. For example the average number of packets in the system, the average duration of a packet being processed,...

2 Implementation

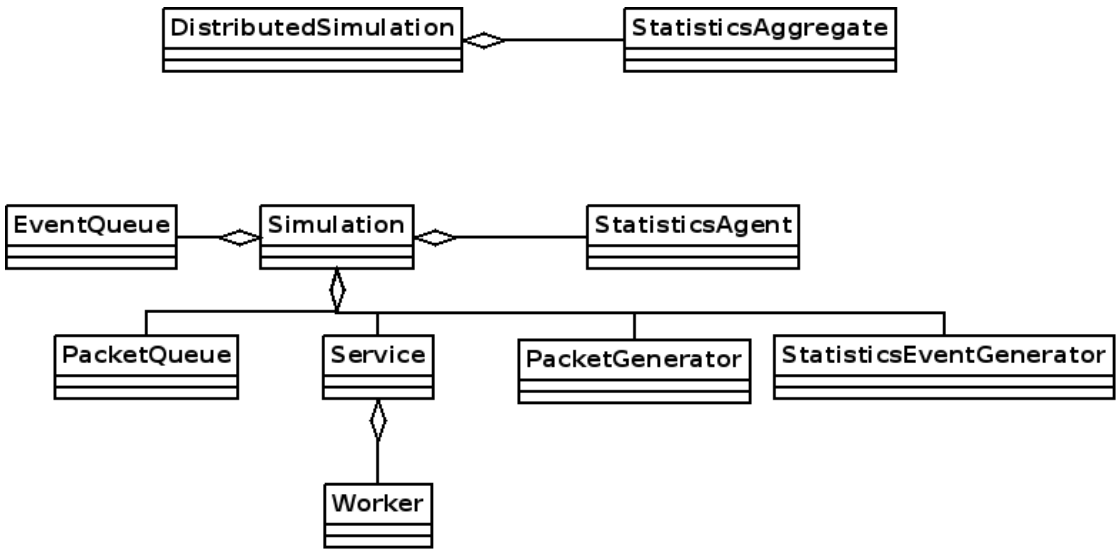
The simulation is basically a single loop which processes the next event. There are three types of events: *new packet*, *packet processed*, *gather statistics*. These events are stored in a priority queue, ordered by time. The packet queue has some logic to decide in which order the packets are passed to the service and the service has some logic to assign a given packet to a worker.

A worker adds a *packet processed* event when given a new packet and the packet generator adds a *new packet* event when generating a packet. Thereby the simulation can simply be started by initializing the event queue with a *new packet* event at time 0.

For a *new packet* event fetch the new packet from the generator and try to give it to the service. If the service is unavailable try the packet queue. If even the packet queue cannot take the packet, drop it.

For a *packet processed* event free the worker and look for available packets in the queue.

This loop is run for a fixed number of events, currently 10000000, before the conditions are checked. The condition is that the standard deviation for all collected statistics is below the specified threshold.



2.1 Parallelization

The given design, which was provided by the lecture, is not well suited to parallel execution due to being a single loop with each iteration depending on the previous ones.

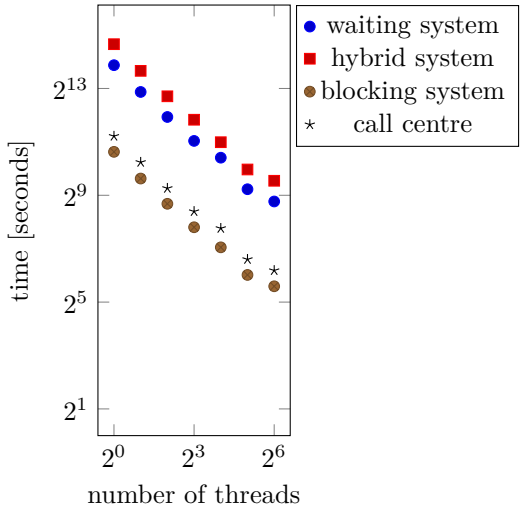
Therefore many simulations were run in parallel and their statistics were combined. The whole system was implemented in C++11 using std::thread and the statistics aggregator was internally synchronized.

3 Benchmark

The simulation was run for four different configurations

Figure 1: time [seconds]

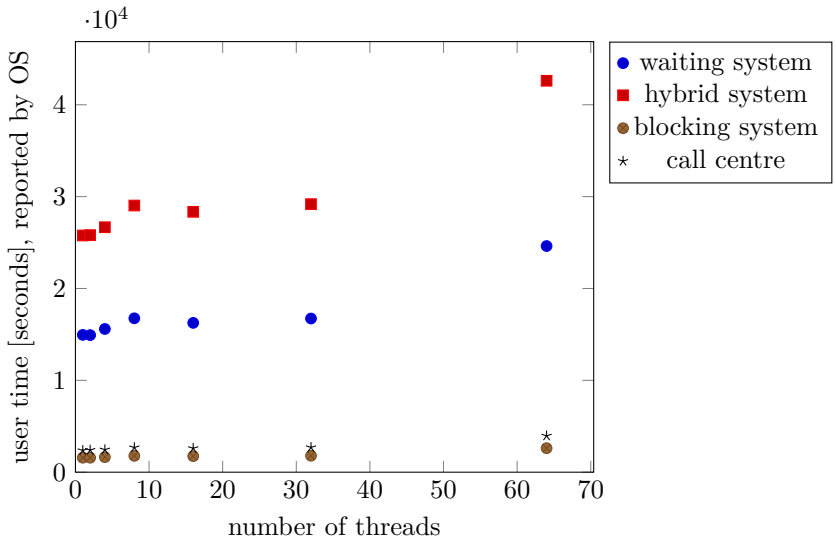
NUMBER OF THREADS	WAITING SYS.	HYBRID SYS.	BLOCKING SYS.	CALL CENTRE
1	14,974.922	25,815.177	1,579.390	2,372.117
2	7,472.907	12,925.881	790.063	1,205.490
4	3,904.172	6,677.185	410.196	613.554
8	2,097.585	3,634.543	223.088	336.229
16	1,356.948	2,028.008	132.484	217.070
32	598.722	999.219	64.740	97.000
64	435.553	745.627	48.220	72.711



The system scales quite well but hyperthreading does not offer a lot of improvement.

Figure 2: user time [seconds], reported by OS

NUMBER OF THREADS	WAITING SYS.	HYBRID SYS.	BLOCKING SYS.	CALL CENTRE
1	14,948.875	25,769.516	1,576.548	2,367.827
2	14,921.622	25,810.758	1,576.906	2,406.941
4	15,591.875	26,667.837	1,637.663	2,449.262
8	16,752.530	29,031.621	1,779.876	2,682.939
16	16,255.411	28,346.498	1,747.671	2,598.014
32	16,728.650	29,181.093	1,791.254	2,693.576
64	24,619.253	42,614.074	2,613.328	3,955.683



The total time used on the system stays relatively stable. Some noise comes from the fact, that there are random events in the simulations. Again, hyperthreading does not give much benefit but counts as normal CPU time.

Figure 3: speed up

NUMBER OF THREADS	WAITING SYS.	HYBRID SYS.	BLOCKING SYS.	CALL CENTRE
1	1.000	1.000	1.000	1.000
2	2.004	1.997	1.999	1.968
4	3.836	3.866	3.850	3.866
8	7.139	7.103	7.080	7.055
16	11.036	12.729	11.921	10.928
32	25.011	25.835	24.396	24.455
64	34.381	34.622	32.754	32.624

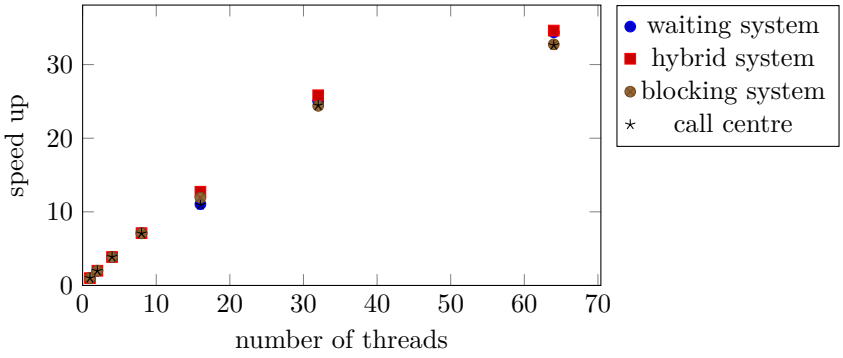
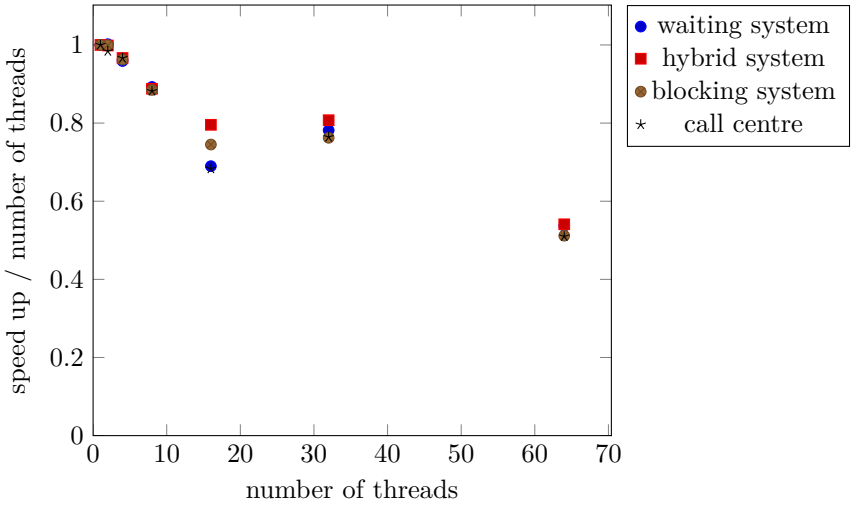


Figure 4: speed up / number of threads

NUMBER OF THREADS	WAITING SYS.	HYBRID SYS.	BLOCKING SYS.	CALL CENTRE
1	1.000	1.000	1.000	1.000
2	1.002	0.999	1.000	0.984
4	0.959	0.967	0.963	0.967
8	0.892	0.888	0.885	0.882
16	0.690	0.796	0.745	0.683
32	0.782	0.807	0.762	0.764
64	0.537	0.541	0.512	0.510

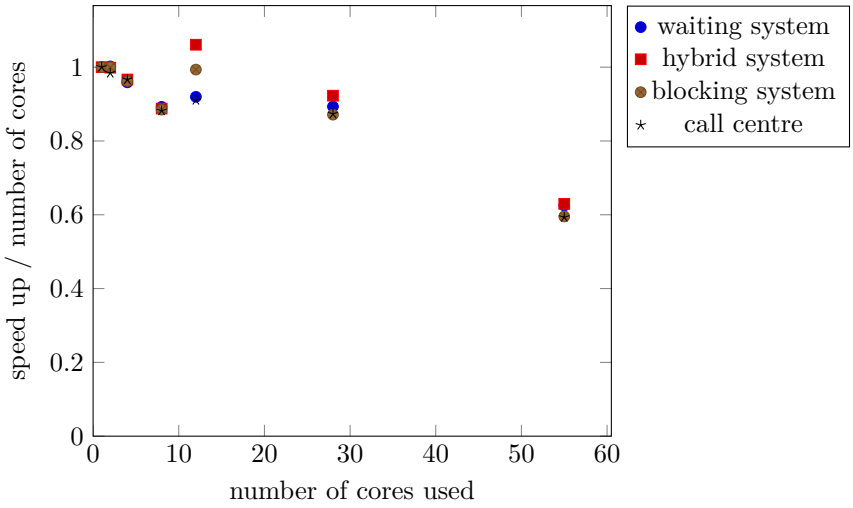


Just looking at the raw output would give a puzzling jump between 16 and 32 threads.

Watching CPU utilization while running the benchmark showed, that the system did not schedule the threads on all available cores. For 1,2,4,8 threads each thread got its own core. But the 16 threads got only 12, the 32 only 28 and the 64 only around 55 (difficult to account right due to the hyperthreading). Adjusting the *speed up / number of threads* to *speed up / number of cores* gives these results:

Figure 5: speed up / number of cores

NUMBER OF CORES	WAITING SYS.	HYBRID SYS.	BLOCKING SYS.	CALL CENTRE
1	1.000	1.000	1.000	1.000
2	1.002	0.999	1.000	0.984
4	0.959	0.967	0.963	0.967
8	0.892	0.888	0.885	0.882
12	0.920	1.061	0.993	0.911
28	0.893	0.923	0.871	0.873
55	0.625	0.629	0.596	0.593



Running more threads than available cores increases efficiency, probably because another thread can run while the current thread is blocked. Therefore achieving a speed up of around 25 on 28 cores is decent.

4 Further remarks

Interesting options that could not be evaluated due to limited time: non-blocking messages or cascaded aggregation. Both may give a small speed boost.

The system in general scales well. Scheduling on the Linux system may be adjusted to utilize all available cores.

The complete code can be found on Github.