Parallelizing queue simulations

Malte Schledjewski

Fachhochschule Kaiserslautern
University of Applied Sciences

Informatik und Mikrosystemtechnik
Zweibrücken

Introduction

As part of the lecture on advanced communications networks (FKOM) a simulation of different queues/services was to be implemented. The goal was to gather various statistics like, for example, the average number of packets in the system or the average duration of a packet in the system. This simulation was then parallelized to speed up the gathering of statistically significant data.

Simulated system

The system consists of three main parts:

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

Each worker needs a certain amount of time per packet. This time as well as the time between new packets follow a negative exponential distribution.

Parallelization

The given design is not well suited to be parallelized or vectorized due to its single-loop-character with dependencies on the previous iteration. Therefore many simulations are run in parallel each with a statistics agent per observed feature. Theses statistics agent collect data points and then send them as an average of a certain bulk size to the aggregator. The current bulk size is 100 000. The aggreagtor is internally synchronized by a mutex. Each simulation is run in a std::thread.

Source Code

The complete source code is available at https://github.com/MalteSchledjewski/FKOM

Conclusions

32 threads on 28 cores give a 25x speed up which indicates a good scaling behaviour. Interesting things to explore, would be using a different scheduler on Linux, using more threads than cores in general and replacing the current aggregation with nonblocking messages or a cascaded aggregation.

Implementation

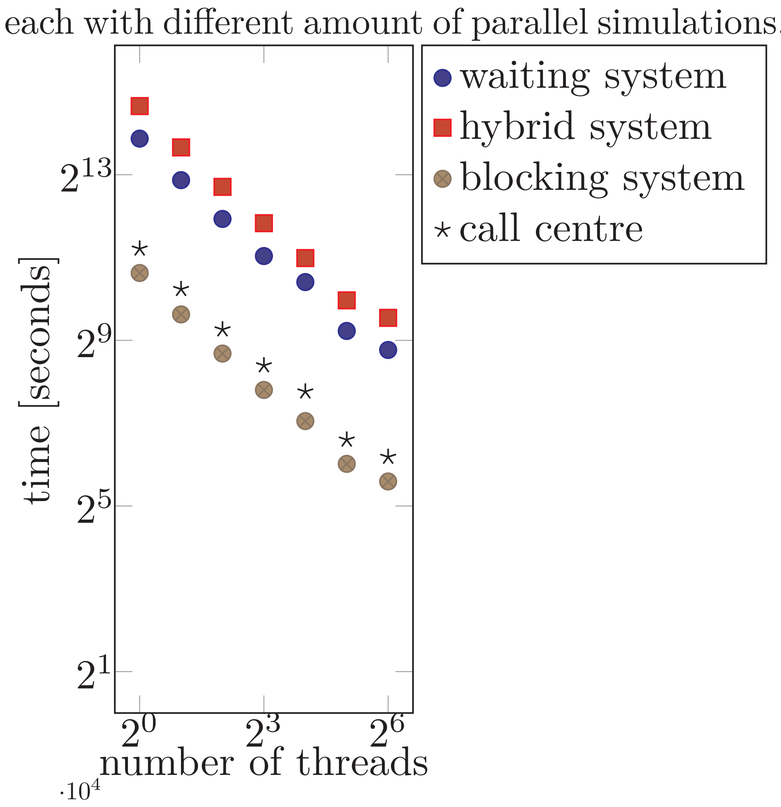
The simulation should be implemented as a discrete event simulation. Possible events are *new packet*, *packet processed* and *gather statistics*. The simulation is started by creating a *new packet* event at time 0.
The service is basically a container of workers with some logic for assigning the packets. The packet queue is just a priority queue with different options of ordering the packets. The event queue is a priority queue sorted by the event's time. Packet generators create a packet and also register the next *new packet* event. The simulation is than basically a loop which always takes the next event from the event queue and then acts on it. If it is a

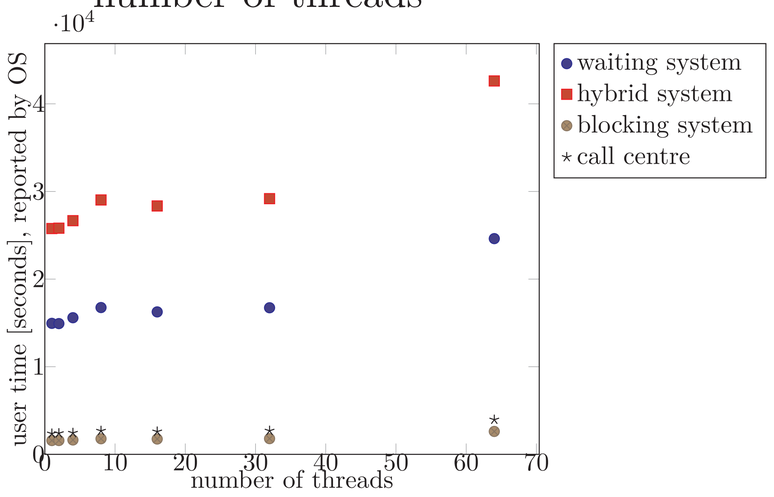
- *new packet* event, it fetches the new packet from the packet generator which in turn registers a new *new packet* event and then tries to give it to the service. If the service is capable of processing the packet right now, the worker is set as busy and a *packet processed* is registered, otherwise the packet queue is checked. If the packet queue cannot hold the new packet, or it displaces an old packet then that packet is dropped.
- *packet processed* event, it frees the worker and checks whether there are pending packets in the packet queue.
- *gather statistics*, several statistics are gathered.

Some statistics concerning packets, like duration in the system, are logged when the packet leaves the system either by being processed or being dropped.
This loop is run for 10 000 000 events before the conditions are checked, that all statistics have a low enough standard deviation. The whole system was implemented in C++11.

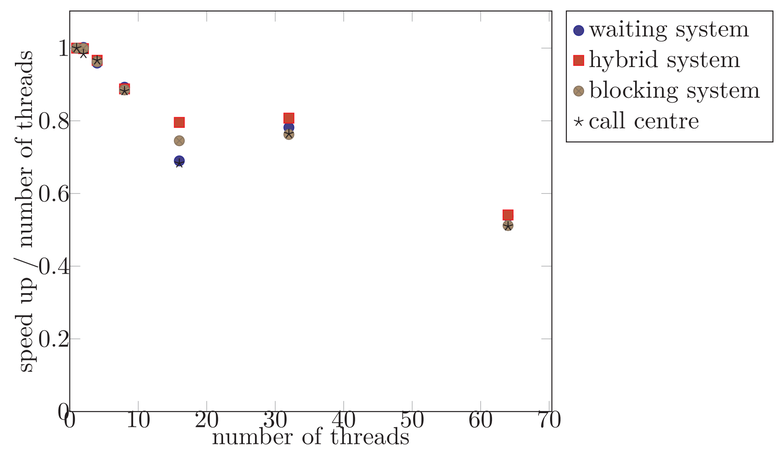
Benchmark

The simulation was run for different scenarios, each with different amount of parallel simulations.

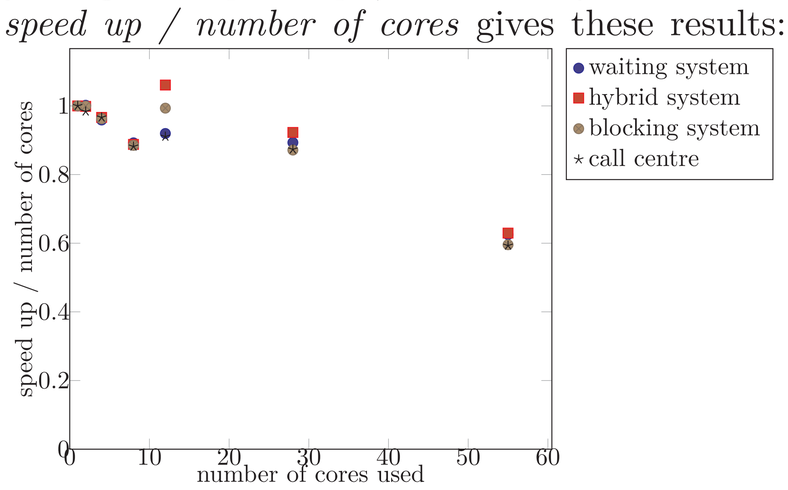




In general the scaling behaviour is well. The total resource usage stays about the same, only hyperthreading does not offer a great benefit.



Looking at the graph there is an interesting jump in efficiency (speed up/ number of threads). But this graph is flawed. There is a mismatch between almost stable resource usage and the drop in speedup vs number of threads. Monitoring core usage showed that the 16 threads got only 12 cores, the 32 only 28 and the 64 only around 55 (difficult to account correctly due to the hyperthreading). Adjusting the *speed up / number of threads* to *speed up / number of cores* gives these results:



This shows that the scaling is even better. Also using more threads than cores improves efficiency probably by letting another thread run, while the current one is blocked.