Report project assignment PDC Summer school 2014

Andreas Karlsson and Niten Olofsson

June 4, 2015

Contents

Abstract

1 Introduction

The software in question, microsimulation (ref to github), implements a probabilistic model, on a person to person level, of the occurence of prostate cancer in men from the age 35 and above. It also includes means of modelling different health care policies (such as screening of population at certain ages), the performance of the different tests utilised, the probability that a person A with symptoms B seeks care themselves, and so forth.

By generating a population, and performing microsimulation (ref), i.e., follow each person (unit) over their life span, and then sort the outcome in different predefined categories, the effect of different policies are evaluated. This fulfills the the criteria of the map-reduce paradigm, where the simulation is the map step, and the sorting into different categories is the reduction step.

The software has been developed "organically" over the years, the first version being implemented in C, then later in C++. It is called from R, using the Rcpp (ref) package. If we can reduce the execution time, and/or increase the sample size, different policies can be evaluated, and better accuracy obtained.

We choose to first adapt the code to use OpenMP to see if any improvements could be achieved, and if time permitted, OpenMP and MPI. The first in order to decrease the

execution time, and the second to allow for much larger sample sizes. In the end, it turned out that we only had time to adapt the code to OpenMP.

2 Overview of the changes to the code

The computation can be modelled using the map-reduce paradigm. First, we analysed the code, took out some parts and made some tests. The conclusion was to first try to parallelise using openmp, then expand to MPI.

First, to the map step of the code, we added

```
#pragma omp parallel shared(nthreads,chunk) private(i,tid,sim,diffTime,cumTime)
...
#pragma omp for schedule(static,1).
```

However, the reduce step was not straight forward to parallelise. In the serial version of the code, an associative array with scope throughout the whole module (*source file*) was used, and every result was incrementally added to the end result. In this reduction step several functions call was made updating the object, where the functions accessed it in the scope of the file and not by a passed reference. So in the first version, we just added

```
#pragma omp critical
  {
  // updating the result, i.e., ``reduction step''
    report.add(FullState(state, ext_grade,
                dx, psa>=3.0, cohort), msg->kind,
                previousEventTime, now());
  }
```

around the section accessing the resource.

After some performance analysis, we realised that the reduction step as implemented became the bottle-neck of the whole software. We then realised that it was possible to let each thread/process build up its contribution to the resulting associative array in a private variable, and then merge these partial results into the shared result instance.

The R software also have a feature of using multi-cores, and we used that one as a benchmark what we should at least obtain

Summarising the steps, we call them

1. naive OpenMP (paralleising the map step)
2. improved OpenMP (first a thread-wise partial reduction, then reducing these partial results)
3. R automatic multi-threading.

3 Performance

From assignment: Prioritize measurements and analysis/interpretation!

Demonstrate use of tools (profiling, ...) , and simple performance model.

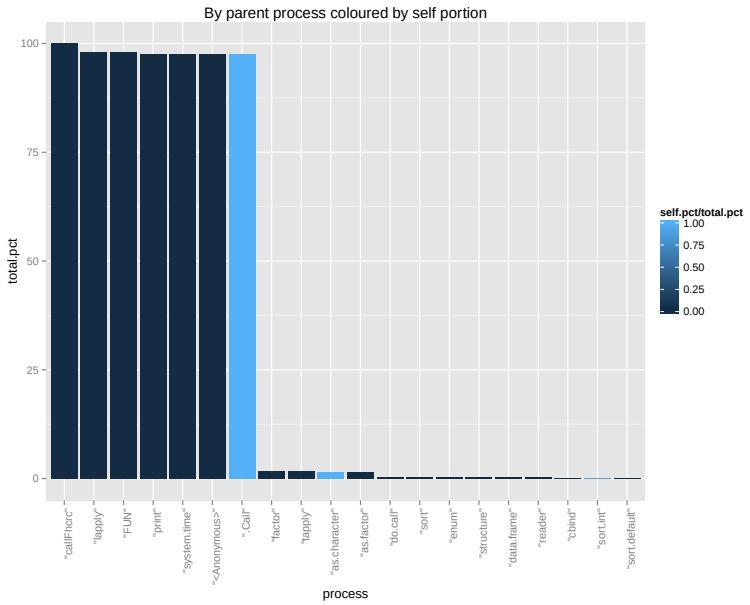

Figure 1: Performance testing on the R side, where the ".Call" is the C++-code which can be run in parallel.

3.1 Measurements

3.2 Profiling

4 Results

4.0.1 Motivational R-side measurement

To motivate the need and choice to go parallel Figure 1 shows the processes from the R-side where ".Call" is the part which is implemented in C++ and which can be run in parallel.

4.0.2 Motivational C++-side measurement

Figure 2 is showing profiling with Valgrind of the main C++ function. This function is called once for each simulated individual and is possible to run in parallel. Noteworthy is that about half of the time is spent in the *EventReport* subfunction. This subfunction is reducing the information from the individuals in the simulation to categories and stores that in a dynamically allocated associative array.

4.1 Measurements of the three parallelisation implementations

Figure 3 shows how the three different implementations of parallelisation scales with additional cores. The *R-side parallelism* and *Improved openMP* scales well with comparable results. The *Naive openMP* implementation with the *EventReport* mentioned in 2 within *#pragma omp critical* statements.

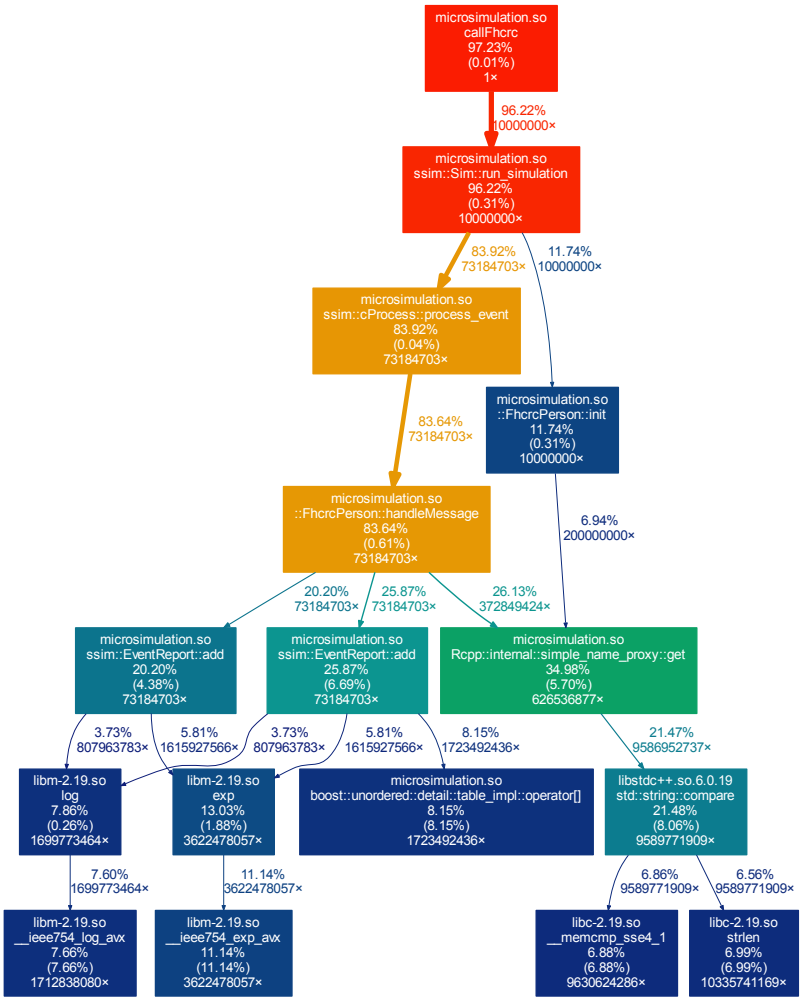

Figure 2: Using Valgrind to profile the main C++ function, which is possible to run in parallel.

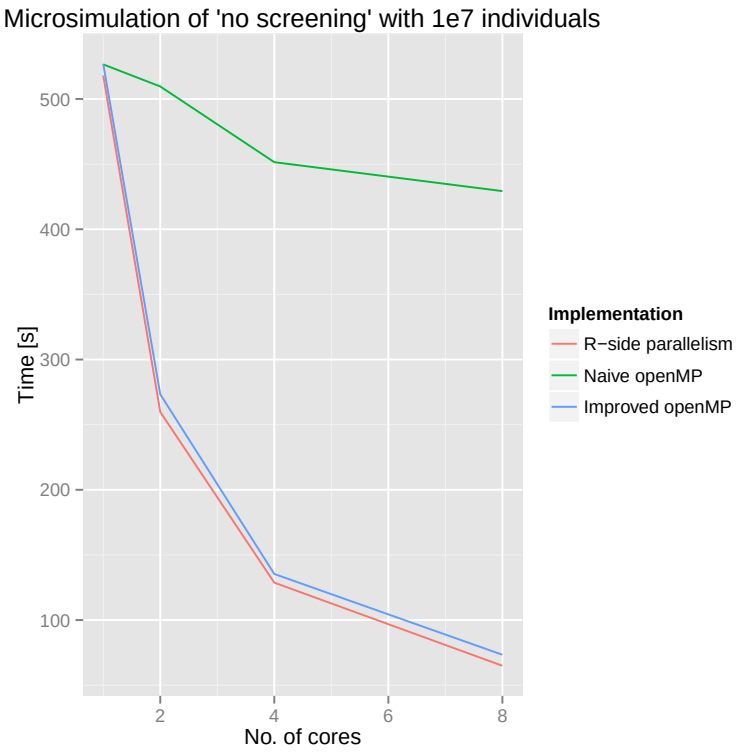

Figure 3: implementations...

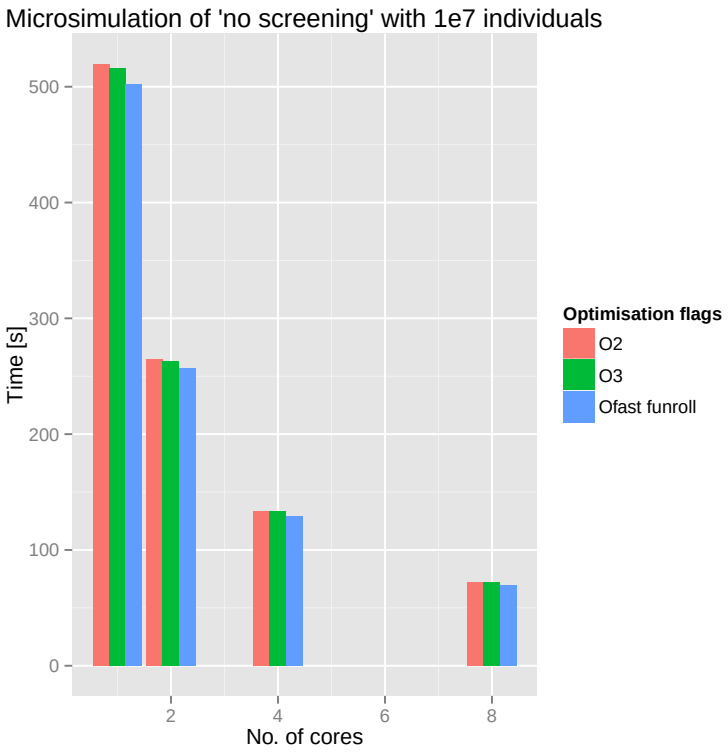

Figure 4: optimisation flags...

4.2 Hybrid openMP and MPI

5 Appendix

5.1 Baseline measurements C++

5.2 Simple approach with OpenMP

Here the simulation loop is run in parallel whereas the data output and some post-processing is run within a omp critical statement.

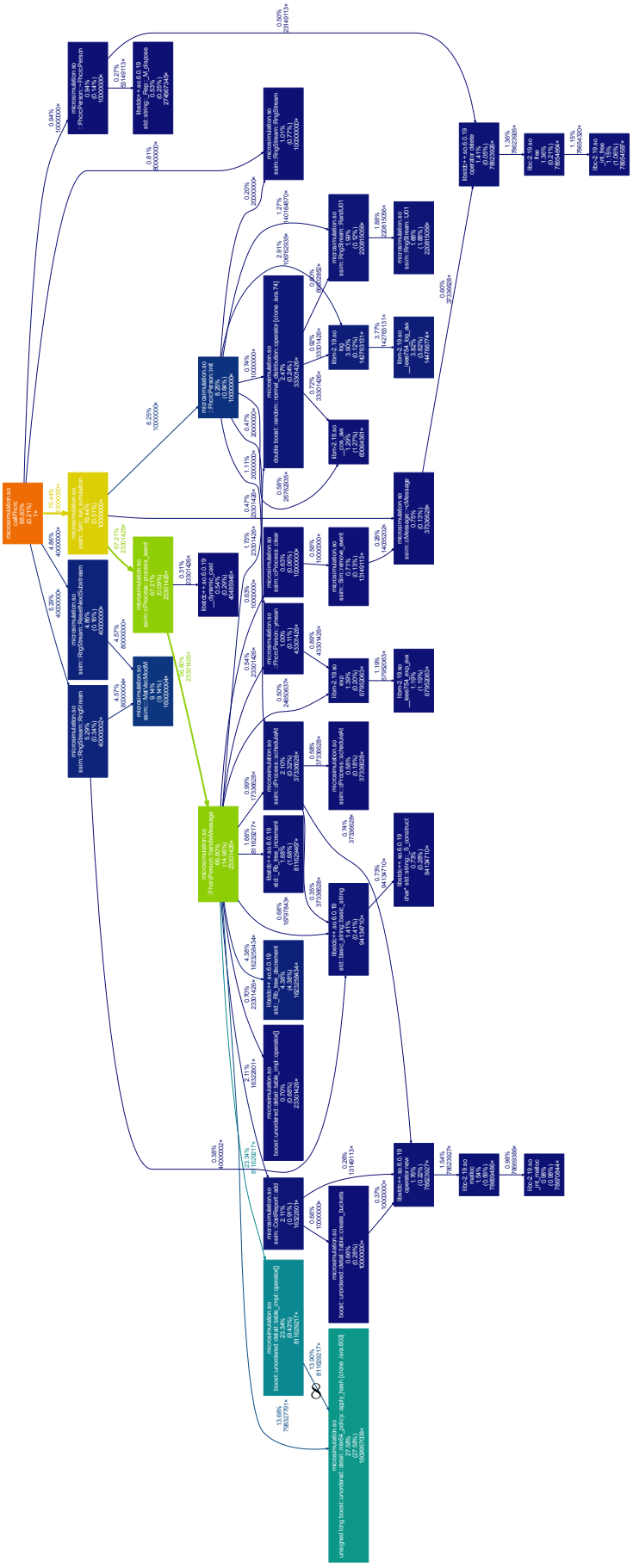

Figure 5: Valgrind measurements at baseline

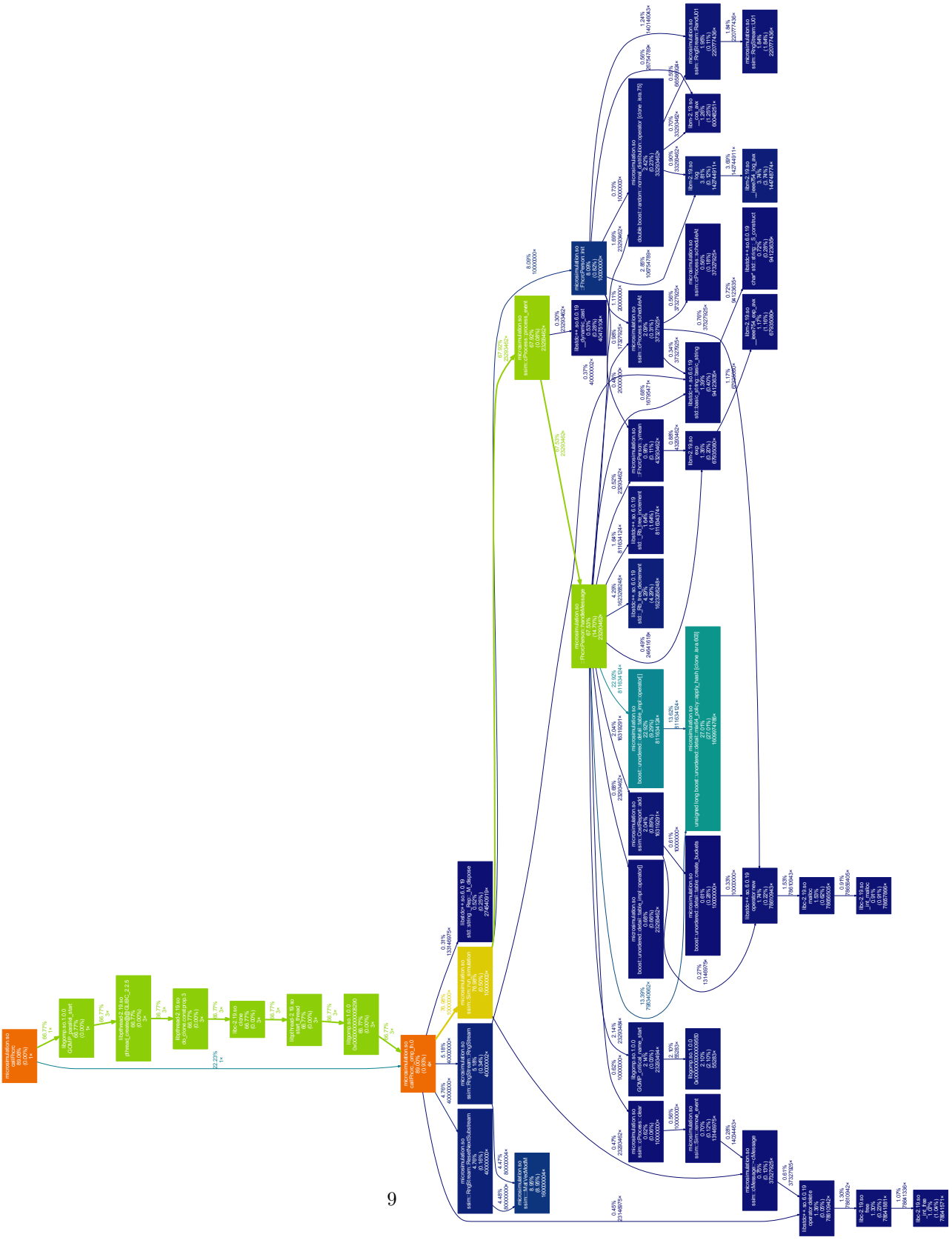

Figure 6: Valgrind results of the naive openMP implementation