



SOCIETY OF ACTUARIES

Article from:

CompAct

January 2008 – Issue 26

Parallel Computing on Multi-Processor Computers Using Freely Available Tools

N. D. Shyamal Kumar

Increasing computational prowess is driving the demand for more realistic modeling and is also partly responsible in regulation becoming less simplistic. But this increasing prowess in the future is going to be delivered by increasing the number of processing units rather than by increasing the clock speed. This trend is already seen with new workstations usually having two or more computing cores. Hence it is imperative that quants acquire parallel computing skills. In this article we show that it takes little to write parallel code to implement embarrassingly parallel algorithms, which is exciting given the prevalence of problems yielding to such algorithms. In one such problem that we discuss here, to our surprise, on a dual core processor we were able to get a speedup greater than two—close to 2.6 in fact! And all of this using only freely available tools for the Windows® operating system.

Programming Paradigm: In this article we will constrain ourselves to shared memory parallel computing, i.e., parallel computing where all threads have access to the same shared memory. The discussion of distributed memory parallel computing, the paradigm for grid computing, will be left for another article. Moreover, we will further restrict our attention to the use of the OpenMP API, which is based on the fork and join execution model. This execution model, see Figure 1, is one where the main thread spawns out multiple worker threads to simultaneously execute sections of code that can be run in parallel, thus speeding up the overall computation. This will be made clearer below when we discuss the implementation of the solution to our problem. The OpenMP site is at <http://www.openmp.org> where one can find useful tutorials (especially, see [3]), list of books on OpenMP, etc.

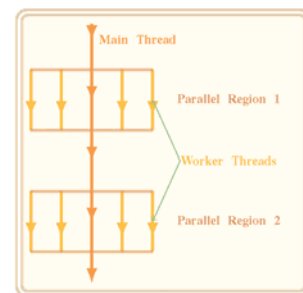


Figure 1: Fork and Join Model

Actuarial Problem: The computational problem we consider is the calculation of the break-even survivorship benefit for the disability product from the Danish market considered in [1]. The age-to-65 product is sold to healthy (able) insureds with an annual premium rate (assumed to be paid continuously) of DKK 10,000, with a premium waiver while the insured is disabled (or invalid), a death benefit at time t defined to be the benefit reserve at that time for a healthy insured, and a survivorship benefit of S payable at the end of the term of the product. Figure 2 depicts the multi-state model underlying the product with disability rate $\sigma(\cdot)$ and mortality rates $\mu(\cdot)$ and $\nu(\cdot)$ for the able and disabled, respectively. The motivation for this product with a strong savings element was to enable insurance companies to compete with banks and other savings institutions, which were only allowed to sell products with an element of insurance (the waiver of premium is included in this product for this sole purpose). This product has didactic value as many companies did not price the product using the correct reasoning, see [1]. The approach taken in [1] is that of using Thiele's differential equation to derive a double-integral expression for S . Here we will adopt a different approach which will lead to a simple Monte-Carlo solution.



*N.D. Shyamalkumar
ASA, an assistant
professor of
statistics &
actuarial science
at the University of
Iowa. He can be
contacted at
shyamal-kumar
@uiowa.edu*

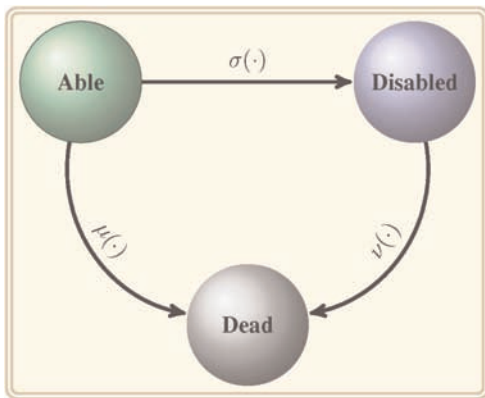


Figure 2: Multi-State Model Underlying the Product

<http://www.iro.umontreal.ca/~lecuyer/myftp/streams00/c>. Since the hazard rates pertain to the Makeham distribution, we used the algorithm as given in [2] to simulate the random occupation time in each state. This algorithm uses two independent exponential variables, and we generated these from uniforms using the log transform.

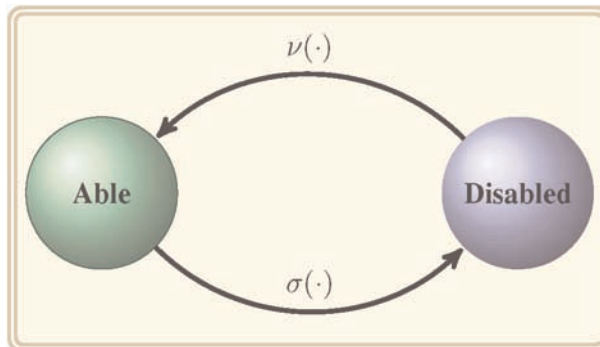


Figure 3: The Reduced Multi-State Model Driving the Algorithm

Figure 2: The Multi-State Model Underlying the Product

Embarrassingly Parallel Algorithm: It is not difficult to see that the survivorship benefit S in the above problem is the same as in the following problem: consider an age-to-65 product sold on a healthy (able) insured for a premium at the annual rate of DKK 10,000, carrying a survivorship benefit of S and a waiver of premium while the insured is disabled; Disability rate is $\sigma(\cdot)$ (as above), and unlike the original product the rate of recovery is taken to be $\nu(\cdot)$ and mortality is ignored (see Figure 3). Since mortality is ignored, S is equal to the expected value of the accumulated premiums. This characterization of S lends itself to Monte-Carlo simulation of S . Hence we will be able to achieve a speedup almost equal to the number of processors (i.e., by dividing the number of simulation runs equally among the processors). Algorithms achieving such speedups are termed embarrassingly parallel.

Caveat for Simulation in Parallel: Unlike sequential simulation where independence of the pseudo-random numbers is more or less guaranteed, one has to take extra care to ensure this in the parallel setting. For one, often pseudo-random number generators are by default initialized using the system clock and this could make many if not all of the simulations carried out on different processors to be identical. A well tested C package that helps to create and easily manage independent streams is RNGStreams, written by Pierre L'Ecuyer. For details and to download the library, see the webpage at

Comments on the Code and Tools: We decided to program in C (see listing) and use the MinGW (Minimalist GNU for Windows) packaged GNU compilers, see <http://www.mingw.org>.

For code using the OpenMP API one will also need the pthreads library which can be found at <ftp://sourceware.org/pub/pthreads-win32/prebuilt-dll-2-8-0-release/>. The OpenMP paradigm allowed us to write parallel code in such a way that it compiles as a sequential program without the `-fopenmp` compiler option. The C code of this project is available on request from the author.

The sequential part of the code is comprised of the lines of code excepting the pragma compiler directives and the code in between `#ifdef` and `#endif` pre-processor conditionals. We observe that the key lines of this part of our code are those from lines 50-69. These simulate the accumulated amount of premiums paid by a single insured. This is repeated for NRUNS number of insureds in order to reduce estimation error.

Now, moving on to the parts of the code that help parallelize the simulation, we notice that

(continued on page 10)

between the first pre-processor conditionals we include the OpenMP library and between the second we set the number of threads. The first pragma directive on line 39 implies that the code between lines 40-70 will be run by each thread, and that all threads will be sharing the global variable `delta` while keeping private copies of all the variables declared within the section. Moreover, the reduction clause creates for each thread its own copy of `mean` (resp., `var`) for the duration of the parallel section, and at the end of the parallel section, unlike private variables which are simply released, the values in these copies will be merged (in our case added) and stored in the global variable `mean` (resp., `var`).


The call to the function `RngStream.CreateStream` on line 44 ensures that each thread has its own independent pseudo random number generator, and that these generators are mutually independent of each other. In order to achieve these goals the function `RngStream.CreateStream` uses a static variable to store its state, and hence is not (and cannot be) thread safe. For this reason, this function call is encapsulated in a pragma `omp critical` directive that allows a thread at a time to make the function call. The last pragma directive on line 48 splits the range of the loop index equally among the threads (load balancing options are available too), and this is the part that achieves a speedup equal to exactly the number of processors.

Results: The computations were done on an Intel® Core™ 2 Duo Processor E6600 (2.4GHz, 4MB shared L2 Cache) box with 4GB RAM and running the Windows Vista™ Enterprise OS. The value of `S` was estimated with a relative error less than 10 basis points by simulating the accumulated premiums for 100 million insureds. The code was run five times with each value for the number of threads listed in Table 1, wherein we report the observed mean running times. It was expected that with two threads we will get a speedup of close to, but less than, two. The

surprises in Table 1 were that we achieved a final speedup far greater than two (likely due to threading allowing more efficient usage of different units of Core™ 2 Duo), and that this required using many more than two threads. In summary, we have found that it is easy to implement embarrassingly parallel algorithms using the OpenMP API, solely employing tools freely available on the Web, with the reward being a speedup by a factor close to the number of processors—or possibly even greater as in the case of Intel® Core™ 2.

Num. of Threads	Time in Seconds	Speedup
1	134.52	1
2	67.41	1.996
4	60.10	2.238
8	55.68	2.416
16	53.38	2.520
32	52.58	2.559
64	52.17	2.578
128	51.94	2.590
256	51.87	2.593

Table 1: Effect of Number of Threads on Computational Time

Acknowledgement: I thank Luke Tierney and Kate Cowles for their enjoyable course on High Power Computing (at the U. of Iowa) which exposed me to many of the technologies discussed in this article. Also, I thank the actuarial science students in my topics course whose interest motivated me to bring this article to its final form. 

References

- [1] Ramlau-Hansen, Henrik (1990). Thiele's Differential Equation as a tool in Product Development in Life Insurance, *Scandinavian Actuarial Journal*, 1990:97-104.
- [2] Pai, Jeffrey S. (1997). Generating Random Variates with a given Force of Mortality and finding a suitable Force Of Mortality by Theoretical Quantile-Quantile Plots. *Actuarial Research Clearing House*, 1997(Vol. 1), 293-312.
- [3] van der Pas, Ruud (2005). An Introduction to OpenMP. Available at http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf

C Code to Calculate the Endowment Amount

```

#include <stdio.h>    /* Standard C input/output library */
#include <math.h>     /* C library for math functions like log, pow etc.. */
#include <hr_time.h>  /* hr_time.c - Support for Timing; http://cplus.about.com/od/howtodoth-
ingsin1/a/timing.htm */
/*
* RNGStreams - Library for Multiple Streams of Random Numbers.
* http://www.iro.umontreal.ca/~lecuyer/myftp/streams00/c/
*/
#include "RNGStream.h"
#include "SRandom.h"  /* Provides rmakeham for Generating from Makeham */
#ifdef _OPENMP
    #include <omp.h>    /* OpenMP Library */
#endif

#define AGE 30
#define TERM 35
#define INTEREST 0.045 /* Annualized Interest */
/* Makeham Hazard Rate: A + B * exp(C*t) */
    #define AMU 0.0005    /* MU - Hazard Rate for Mortality */
    #define CMU 0.08749823558
    #define ASIGMA 0.0004 /* SIGMA - Hazard Rate for Mortality */
    #define CSIGMA 0.1381551353
    #define BSIGMA 0.00021877616 /* pow( 1 0 . 0 , ( 0 . 0 6*AGE-5.46) ) */
    #define BMU 0.001047128548051 /* pow( 1 0 . 0 , ( 0 . 0 3 8*AGE-4.12) ) */
#define NRUNS 100000000 /*No. of Runs*/

int main()
{
    double delta=log(1.0+INTEREST), mean=0, var=0;

#ifdef _OPENMP
    int nthreads;
    printf("Please Enter the Number of Threads You Wish to Use: ");
    scanf("%d",&nthreads);
    omp_set_num_threads(nthreads);
    printf("The Number of Threads Used: %i\n", nthreads);
#endif

    stopWatch t;
    startTimer(&t);

#pragma omp parallel default(none) shared(delta) reduction(+:mean,var)
    { /* Parallel Section */
        RngStream g;
#pragma omp critical
        {

```

```

        g = RngStream_CreateStream (""); /* One Thread at a Time */
    }
    int j;
    double time_to_term, tsigma, tmu, bmu, bsigma, premium;
    #pragma omp for
    for (j=0;j<NRUNS;j++) {
        premium=0;
        time_to_term=TERM;
        tsigma=0;
        tmu=0;
        bmu=BMU;
        bsigma=BSIGMA;
        /* Generate the Premium Paid by a Single Insured*/
        while (time_to_term>0.00000001){
            /* Insured in Able State*/
            bsigma=bsigma*pow(10.0,(0.06*(tmu+tsigma))); /* Shift Hazard Rate Sigma */
            tsigma=rmakeham(&g, ASIGMA, bsigma, CSIGMA); /* Generate Time in Able State */
            if (tsigma> time_to_term) tsigma=time_to_term;
            premium=premium+pow(1.0+INTEREST,time_to_term)-pow(1.0+INTEREST,time_to_term-tsigma);
            time_to_term=time_to_term-tsigma;
            if (time_to_term>0) { /* Insured in Disabled State */
                bmu=bmu*pow(10.0,(0.038*(tmu+tsigma))); /* Shift Hazard Rate Mu */
                tmu=rmakeham(&g, AMU, bmu, CMU); /* Generate Time in Disabled State */
                if (tmu> time_to_term) tmu=time_to_term;
                time_to_term=time_to_term-tmu;
            }
        }
        /* Increment the sum and sum of squares */
        mean=mean+premium;
        var=var+premium*premium;
    }
} /* End of Parallel Section */

mean=10*mean/NRUNS/delta; /* Point Estimate of the Endowment Amount in Thousands*/
var=(100*var/NRUNS/(delta*delta)-mean*mean)/NRUNS; /*Its Estimated Variance */
stopTimer(&t);

printf("Elapsed Time: %g \n", getElapsedTime(&t));
printf("A Point Estimate: %6.3f\n", mean);
printf("Its Std. Error: %10.8f\n", sqrt(var));
printf("A 99.9%% CI: ( %6.3f,%6.3f)\n", mean-3.090232*sqrt(var), mean+3.090232*sqrt(var));
}

```