A Parallelised High Performance Monte Carlo Simulation Approach for Complex Polymerisation Kinetics

Supporting Information Section

Hugh Chaffey-Millar,[a] Don Stewart,[b]
Manuel M. T. Chakravarty,[b] Gabriele Keller,[b,*] Christopher Barner-Kowollik[a,*]

[a]*Center for Advanced Macromolecular Design, School of Chemical Sciences and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia, c.barner-kowollik@unsw.edu.au*

[b]*Programming Languages and Systems, School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia, keller@cse.unsw.edu.au*

1 Miscellaneous code details

Program code can be downloaded from `http://www.camd.unsw.edu.au/parapolysim`

1.1 Reactions that the code is capable of simulating

		A	$\rightarrow$	B	$+$ C		(initiator decomposition)
A	$+$	B	$\rightarrow$	$P(1)$			(initiation)
$P(i)$	$+$	M	$\rightarrow$	$Q(i+1)$			(propagation)
$P(i)$	$+$	A	$\rightarrow$	$Q(i)$			(addition, no chain len. alteration)
		$P(i)$	$\rightarrow$	$Q(i-1)$	$+$ M		(de-propagation)
		$P(i)$	$\rightarrow$	$Q(i)$	$+$ B		(de-addition)
$P(i)$	$+$	$Q(j)$	$\rightarrow$	$R(i)$	$+$ $S(j)$		(disproportionation)
		$A(i)$	$\rightarrow$	$B(j)$			(change)
$P(i)$	$+$	$Q(j)$	$\rightarrow$	$R(i+j)$			(combination)
$P(i)$	$+$	$Q(j)$	$\rightarrow$	$\mathbf{C}(i,j)$			(complex formation)
$P(i)$	$+$	$\mathbf{C}(j,k)$	$\rightarrow$	$\mathbf{C}(i,j,k)$			(complex addition)
		$\mathbf{C}(i_1,i_2\ldots i_n)$	$\rightarrow$	$D(i_1+i_2+\ldots+i_n)$			(consolidation 1)
$\mathbf{C}(i_1,i_2\ldots i_n)$	$+$	$P(j)$	$\rightarrow$	$D(i_1+i_2+\ldots+i_n+j)$			(consolidation 2)
$\mathbf{C}(i_1,i_2\ldots i_n)$	$+$	$\mathbf{D}(j_1,j_2\ldots j_n)$	$\rightarrow$	$E(i_1+i_2+\ldots+i_n+j_1+j_2+\ldots+j_n)$			(consolidation 3)
$\mathbf{C}(i_1,i_2\ldots i_n)$	$+$	M	$\rightarrow$	any of $\mathbf{C}(i_1+1,i_2\ldots i_n)$, $\mathbf{C}(i_1,i_2+1\ldots i_n)$ etc...			(complex propagation)
		$\mathbf{C}(i,j)$	$\rightarrow$	$P(i)$	$+$ $Q(j)$		(complex scission)
		$\mathbf{C}(i_1,i_2\ldots i_n)$	$\rightarrow$	$\mathbf{C}(i_1,i_2\ldots i_{n-1})$	$+$ $P(i_n)$		(complex scission)

1.2 Random number generation

Two sources of random numbers were tested: (a) the Mersenne Twister (MT) random number generator[1] and (b) randoms from www.randoms.org. These two methods lead to slight differences in the speed of simulation with random numbers read from disk yielding a speed increase of approximately 10% in the simple test model. The MT random number generated has been used before in the simulation of polymerisation kinetics,[2] as has the method of reading numbers from disk.[3] Both are equally valid as long as the pool of random numbers stored on disk is large. In the current research, all benchmarks rely on random numbers generated using the MT random number generator since this method was deemed to be more general.

40 Mb of random bits were downloaded from random.org (but could in principal have come from any random source). These bits were used as 2 byte positive integers, γ_i, in the range [0,$2^{16} - 1$] and were subsequently converted into random floats, γ, in the range (0,1) using the conversion $\gamma = (1 + \gamma_i)/(2^{16} + 1)$, resulting in 80 Mb of single precision random numbers with six significant figures. When these were used in a simulation, the entire 80 Mb file was read into memory at the commencement of program execution. The time taken to read these from disk (normally of the order of several seconds) was not included in the measured simulation time.

1.3 Compilation details

For compilations using the GNU Compiler Collection (GCC), the performance related flags for Intel Pentium 4 systems were

```
-O3 -fomit-frame-pointer -ffast-math -msse3 -march=pentium4
```

and for Advanced Micro Devices (AMD) Athlon systems were

```
-O3 -fomit-frame-pointer -ffast-math -march=pentium3
```

and for AMD Opteron systems were

```
-O3 -fomit-frame-pointer -ffast-math -march=opteron
```

For compilations using the Intel C++ Compiler (ICC), the performance related flags were

```
-fast -fomit-frame-pointer
```

for Intel Pentium 4 and

```
-O3 -static -ipo -fomit-frame-pointer -march=pentium3
```

for AMD Athlon. Profiling experiments were only conducted on the Pentium 4 architecture using the ICC, and these used the switches

```
-O3 -static -ipo -fno-inline -p
```

and the analysis of the raw output was performed using the tool `gprof`.

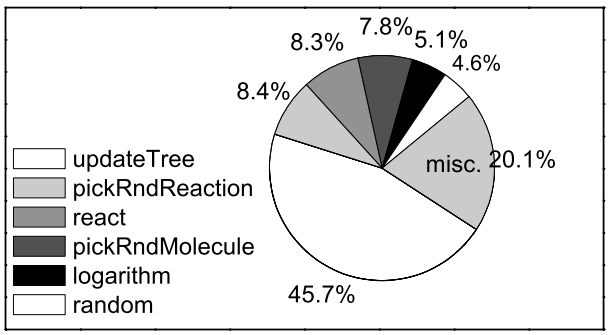

Figure 1: Profiling results of the Monte Carlo code for simulation of the complex test system to 100 seconds using 10^{10} particles. It indicates the time consumed by various parts of the code: updateTree() updates the reaction probability tree, pickRndReaction() chooses a reaction to perform based on its probability, react() manages a particular function at a high level by calling other functions such as pickRndReaction(), pickRndmolecule(), etc. as required; pickRndMolecule() randomly chooses a molecule; logarithm() refers to the built-in log function in the standard math library; random() generates random integers using the Mersenne Twister algorithm; misc. corresponds to other operations such as memory copying. This profile was generated with communication turned off. The relevant excerpt of the profiling output is contained in the supplementary data section.

1.4 Structure of communicated data

A typical package of data used to communicate the state between various nodes during parallel operation appears below. S is the number of species,

description	header	S species counts	S max. chain lens.
layout	node time, monomer auditing information, error flags	$N_0 \ldots N_{S-1}$	$L_{\text{max},0} \ldots L_{\text{max},S-1}$

continued...

S MWDs			Padding
$n_{1,0} \ldots n_{1,L_{\text{max},0}}$	$\ldots$	$n_{S,0} \ldots n_{M,L_{\text{max},S}}$	0, 0, 0. . .

1.5 Profiling

Figure 1 displays profiling results indicating the fraction of the run time consumed by various parts of the program. The task which consumes the greatest amount of time is updating the reaction probability tree. Interestingly, the use of this tree data structure provided a significant improvement in performance over the linear data structure. The reason that this step takes a large amount of time is that whenever a reaction occurs, a number of floating point calculations are required: (a) multiplications to calculate the new rates of all reactions whose reactants have changed in number and (b) a number of additions to update various parts of the tree data structure. It is important to note that

such a graph is approximate since the inclusion of instructions into the executable to generate such data affects the way the speed at which the code runs and the relative time consumption by various functions may be slightly different for a non-profiled executable.

1.6 Example of C code generated by Haskell

This section assumes familiarity with the C programming language. As described in the manuscript proper, the Haskell program analyses a kinetic scheme and generates specialised C code. The following code excerpts from the complex test system demonstrate (A) how when a reaction with index `reactionIndex` is chosen, the specialised C code performs chain length arithmetic and decides which products are formed by a particular reaction and (B) how the program re-calculates the reaction probabilities which have changed and updates the reaction probability tree data structure.

(A) Code excerpt demonstrating control of each reaction

```
switch (reactionIndex) {
     case decomposition:
         no_of_res = 2;         // number of products
         prod1_ind = I_Star;    // product 1 index
         prod2_ind = I_Star;    // product 2 index
          break;
     case decompositionFalse:
         no_of_res = 2;
         prod1_ind = Junk;
         prod2_ind = Junk;
          break;
     case initiation:
         prod1_ind = P;
         prod1_lens[0] = 1;
          break;
     case reInitiation:
         prod1_ind = A;
         prod1_lens[0] = 1;
          break;
     case propagationP:
         prod1_ind = P;
         prod1_lens[0] = react1_lens[0] + 1;  // Demonstrates chain length
                                              // arithmetic for propagation
          break;
     ⋮
     case terminationAA:
         prod1_ind = A_coupled;
         prod1_lens[0] = react1_lens[0]
                       + react2_lens[0];    // Demonstrates chain length
                                            // arithmetic for combination.
          break;
     ⋮
}
```

(B) Code excerpt demonstrating reaction choice tree updates

```
switch (reactionIndex) {
```

```
        case decomposition:
            state.reactProbTree[31] = state.reactions[0].rc * (state.ms_cnts[I]);
            state.reactProbTree[32] = state.reactions[1].rc * (state.ms_cnts[I]);
            state.reactProbTree[33] = state.reactions[2].rc * (state.ms_cnts[I_Star] * state.ms_cnts[M]);
            state.reactProbTree[15] = state.reactProbTree[31] + state.reactProbTree[32];
            state.reactProbTree[16] = state.reactProbTree[33] + state.reactProbTree[34];
            state.reactProbTree[7] = state.reactProbTree[15] + state.reactProbTree[16];
            state.reactProbTree[3] = state.reactProbTree[7] + state.reactProbTree[8];
            state.reactProbTree[1] = state.reactProbTree[3] + state.reactProbTree[4];
            state.reactProbTree[0] = state.reactProbTree[1] + state.reactProbTree[2];
            break;
    ⋮
        case propagationP:
            state.reactProbTree[33] = state.reactions[2].rc * (state.ms_cnts[I_Star] * state.ms_cnts[M]);
            state.reactProbTree[34] = state.reactions[3].rc * (state.ms_cnts[R] * state.ms_cnts[M]);
            state.reactProbTree[35] = state.reactions[4].rc * (state.ms_cnts[P] * state.ms_cnts[M]);
            state.reactProbTree[36] = state.reactions[5].rc * (state.ms_cnts[A] * state.ms_cnts[M]);
            state.reactProbTree[16] = state.reactProbTree[33] + state.reactProbTree[34];
            state.reactProbTree[7] = state.reactProbTree[15] + state.reactProbTree[16];
            state.reactProbTree[17] = state.reactProbTree[35] + state.reactProbTree[36];
            state.reactProbTree[8] = state.reactProbTree[17] + state.reactProbTree[18];
            state.reactProbTree[3] = state.reactProbTree[7] + state.reactProbTree[8];
            state.reactProbTree[1] = state.reactProbTree[3] + state.reactProbTree[4];
            state.reactProbTree[0] = state.reactProbTree[1] + state.reactProbTree[2];
            break;
        case propagationA:
            state.reactProbTree[33] = state.reactions[2].rc * (state.ms_cnts[I_Star] * state.ms_cnts[M]);
            state.reactProbTree[34] = state.reactions[3].rc * (state.ms_cnts[R] * state.ms_cnts[M]);
            state.reactProbTree[35] = state.reactions[4].rc * (state.ms_cnts[P] * state.ms_cnts[M]);
            state.reactProbTree[36] = state.reactions[5].rc * (state.ms_cnts[A] * state.ms_cnts[M]);
            state.reactProbTree[16] = state.reactProbTree[33] + state.reactProbTree[34];
            state.reactProbTree[7] = state.reactProbTree[15] + state.reactProbTree[16];
            state.reactProbTree[17] = state.reactProbTree[35] + state.reactProbTree[36];
            state.reactProbTree[8] = state.reactProbTree[17] + state.reactProbTree[18];
            state.reactProbTree[3] = state.reactProbTree[7] + state.reactProbTree[8];
            state.reactProbTree[1] = state.reactProbTree[3] + state.reactProbTree[4];
            state.reactProbTree[0] = state.reactProbTree[1] + state.reactProbTree[2];
            break;
    ⋮
}
```

2 Monte Carlo profiling – raw output

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
45.65	63.46	63.46	110451220	0.00	0.00	updateTree
8.44	75.19	11.74	110451220	0.00	0.00	pickRndReaction
8.31	86.74	11.55	110451220	0.00	0.00	react
7.75	97.51	10.77	206403007	0.00	0.00	pickRndMolecule_order1
5.10	104.59	7.09				logf.J
4.55	110.92	6.33	337544612	0.00	0.00	genrand_int32
4.04	116.54	5.62				round.J
2.71	120.31	3.77				__intel_new_memcpy
2.52	123.82	3.51	1	3.51	114.33	compute
2.47	127.25	3.43				__profile_frequency
1.87	129.85	2.60	119405038	0.00	0.00	adjustMolCnt_order1
1.73	132.25	2.40				recvmsg
1.39	134.18	1.93	220902440	0.00	0.00	genrand_real3
0.88	135.40	1.22	220902440	0.00	0.00	randomProb
0.87	136.61	1.21	116642172	0.00	0.00	genrand_real1
0.81	137.73	1.12				_intel_fast_memcpy.J
0.19	137.99	0.26				doneit
0.15	138.20	0.21				_intel_fast_memcpy.H
0.12	138.37	0.17				_intel_fast_memcpy
0.11	138.52	0.15				_intel_fast_memcpy.A
0.10	138.66	0.14				logf
0.08	138.76	0.11				round
0.07	138.86	0.10	1	0.10	0.10	compressState
0.04	138.92	0.06				round.A
0.03	138.96	0.04				logf.A
0.02	138.98	0.03	108	0.00	0.00	strAppend
0.01	139.00	0.02	1	0.02	0.02	getConvertedMonomer
0.00	139.00	0.00	988	0.00	0.00	toConc
0.00	139.00	0.00	285	0.00	0.00	name
0.00	139.00	0.00	111	0.00	0.00	takeSome
0.00	139.00	0.00	39	0.00	0.00	conversion
0.00	139.00	0.00	18	0.00	0.00	max
0.00	139.00	0.00	2	0.00	0.01	file_write_state
0.00	139.00	0.00	2	0.00	0.00	init_genrand
0.00	139.00	0.00	2	0.00	0.00	readTimerSec
0.00	139.00	0.00	2	0.00	0.00	startTimer
0.00	139.00	0.00	1	0.00	0.00	initSysState
0.00	139.00	0.00	1	0.00	114.45	main
0.00	139.00	0.00	1	0.00	0.02	monomerAudit
0.00	139.00	0.00	1	0.00	0.00	print_kinetic_model
0.00	139.00	0.00	1	0.00	0.00	print_state
0.00	139.00	0.00	1	0.00	0.00	print_state_summary
0.00	139.00	0.00	1	0.00	0.00	readTimer

3 Kinetic parameters/initial concentrations

rate coefficient/quantity	units	simple test system	complex test system
k_d	s^{-1}	6.1×10^{-6}	1.97×10^{-4}
f		0.64	0.64
k_i	$\mathrm{L\ mol}^{-1}\mathrm{s}^{-1}$	1.7×10^{3}	664
k_p	$\mathrm{L\ mol}^{-1}\mathrm{s}^{-1}$	340	664
k_t	$\mathrm{L\ mol}^{-1}\mathrm{s}^{-1}$	1×10^{8}	1×10^{8}
$k_\mathrm{t,A}$	$\mathrm{L\ mol}^{-1}\mathrm{s}^{-1}$		5×10^{7}
$k_\mathrm{t,A,A}$	$\mathrm{L\ mol}^{-1}\mathrm{s}^{-1}$		1×10^{7}
k_tr	$\mathrm{L\ mol}^{-1}\mathrm{s}^{-1}$	5×10^{5}	
$k_{\beta,1}$	$\mathrm{L\ mol}^{-1}\mathrm{s}^{-1}$		5×10^{5}
$k_{-\beta,1}$	s^{-1}		1×10^{5}
k_β	$\mathrm{L\ mol}^{-1}\mathrm{s}^{-1}$	5×10^{5}	5×10^{5}
$k_{-\beta}$	s^{-1}	1	1×10^{5}
initial $c_\mathrm{initiator}$	$\mathrm{mol\ L}^{-1}$	1×10^{-3}	1.2×10^{-3}
initial $c_\mathrm{RAFT\text{-}R}$	$\mathrm{mol\ L}^{-1}$	1×10^{-2}	5×10^{-3}
initial c_monomer	$\mathrm{mol\ L}^{-1}$	8.3	8.17

4 PREDICI implementation and benchmarking

Miscellaneous information relating to the PREDICI implementation of the models:

- For bimolecular termination reactions of a reactant with concentration c, PREDICI, by default, uses the IUPAC convention for the rate law which requires definition as $dc/dt = -2 \cdot k \cdot c^2$. In PREDICI this is even the case for bimolecular reactions in which the two reactant species are not identical. In order to achieve (correctness and) concordance with the Monte Carlo model, for the bimolecular termination reaction between unlike species, non-IUPAC conformity needed to be selected in the reaction options.

- PREDICI is not capable of simulating species with more than one chain length associated with them, for example, the **Q** species in both the simple and complex test models. In order to represent such species, *chain length memory* reactants must be used in the PREDICI implementation, as described previously.

- The optimised PREDICI simulations used an adaptive accuracy parameter which was low (circa 0.1) for the first second of polymerisation time and subsequently increased to the final value over the next 9 seconds. See manuscript propper for actual accuracy values used.

5 Hardware configuration

Details of the hardware, further to those appearing in the experimental section of the manuscript propper: The Linux machines were running Linux version 2.6.19.7-general and had gcc version 4.0.3 20051201 (prerelease). All PREDICI simulations were run on Windows XP Professional Version 2002.

References

[1] Matsumoto, M.; Nishimura, T. *ACM Trans. Model. Comput. Simul.* **1998**, *8*, 3.

[2] Drache, M.; Schmidt-Naake, G.; Buback, M.; Vana, P. *Polymer* **2004**, *46*, 8483.

[3] Prescott, S. W. *Macromolecules* **2003**, *36*, 9608.