`internals_mc` on multiple cores

dimis

April 3, 2008

1 Introduction

This project focuses on parallelizing a Monte Carlo method on modern multi-core processors with the use of PTHREADS library. A description of the mechanics of the computational problem can be found in section 2. The reader may refer to [5] hosted in [9] for more details about the problem as well as pointers to other related practical problems. The computational goal is to compute the probability distribution of various events given their weights and an underlying model which also influences their appearences. Experimentation results are presented in section 4 indicating the speedup as well as the efficiency achieved in practice on a dual core machine.

2 A general framework for the problem.

There is a group of resources $\mathcal{R}$; $|\mathcal{R}| = \mathsf{N}$. There is a basket $\mathcal{B}$ with $\mathsf{M} < \mathsf{N}$ slots; each slot $\mathsf{s_i}$ is composed by k pockets which is constant for all slots. There is also a weight function $w_{\mathcal{B}} : \mathcal{R} \to \mathbb{N}$, which depends on the basket $\mathcal{B}$. We place resources into pockets, subject to the restriction that all k pockets in the same slot have the same resource. Moreover, different slots contain different resources. Initially l pockets are non-empty (either in the same or in different slots). Now we have the following game. The game is played with rounds. It lasts $(\mathsf{M} \cdot \mathsf{k} - \mathsf{l})$ rounds and at each step we assign a resource to one of the pockets. A sequence of $(\mathsf{M} \cdot \mathsf{k} - \mathsf{l})$ rounds forms an *episode*. At each step (round) there are two options (resources) and the player picks one of them to be added into the basket $\mathcal{B}$. The way the two options are presented to the player is determined by the weight function $w_{\mathcal{B}}$ and some underlying model. For the moment let's forget about the model. The two options, say a and b are presented as follows: a (left option) is a resource that can be found in some slot of the basket, but *not all* k pockets of that slot are full so far, while b (right option) is a resource that can *not* be found in any slot (pocket) of the basket so far. In the case where all k pockets are full for the resources occupying all $\mathsf{j} < \mathsf{M}$ slots, then both a and b are resources that have not been selected so far. In the case where all M slots have filled at least $\mathsf{1}$ pocket, then both a and b are selected among the resources already in the basket for which there is an available pocket.

2.1 The main computational problem.

The user has a strategy (preference) when selecting resources. The computational goal is to determine the probability that each resource has to be in the basket at the end of the game according to user's strategy for various strategies.

2.2 How does the model affect the options.

There are some groups of resources, possibly all with different cardinality. Each group $\mathtt{g}$ has a period $\mathtt{p_g}$. Now, if $\mathtt{p_g} - 1$ steps of the game have been played and no resource $\mathtt{r} \in \mathtt{g}$ has been offered as an option, then on the next step of the game, we have an *exception* and one of the resources in $\mathtt{g}$ will be offered. The probability for each resource $\mathtt{r} \in \mathtt{g}$ to appear is again dependent on $w_{\mathcal{B}}$. If two (or more) exceptions coincide, then, there is a hierarchy function $\mathtt{h}$ which determines which exception will be enforced. All other exceptions are stored as exceptions for the next step.

2.3 The real deal.

The practical problem that gave rise to the above general framework comes from a computer game that is actively played worldwide; *Heroes of Might and Magic III*. The resources are skills that can be obtained by a hero controlled by the user. The basket slots are the available slots for different skills and the pockets reflect the level of expertise of each skill. In most cases it holds $|\mathcal{R}| = \mathtt{N} = 28, \mathtt{M} = 8, \mathtt{k} = 3$, and $\mathtt{l} = 2$, with some minor variations on $\mathtt{k}$ and $\mathtt{l}$ on their starting values. Moreover there are 3 known groups of skills, one of them (WISDOM) containing 1 resource (wisdom) with period $\mathtt{p_1} = 6$, the other (MAGIC) with 4 resources (air, earth, fire, water) with period $\mathtt{p_2} = 4$, and the third (REST) containing 23 resources with $\mathtt{p_3} = \infty$. The sets are disjoint. $w_{\mathcal{B}}(\mathtt{r}) \in \{0, 1, \ldots, 10\}$, for any $\mathtt{r}$ and $\mathcal{B}$.

2.4 Solving the problem.

Apart from some naive strategies followed by the user, most of the strategies, and especially those closer to real human games, imply state spaces that are huge for brute force approaches even by modern computers. In order to counter the curse of dimensionality at least for practical purposes, a Monte Carlo method approach is followed in order to compute the implied probability distributions in these cases. The (serial/single-core) work that has been done so far can be found in [4] which is hosted in [2]. Brute force results on a naive selection strategy can be found in [3]. The results from the approach in [3] are good validators on the estimates that are generated by the Monte Carlo implementations (either the serial one or the parallel one considered here).

3 Working in parallel.

In [1] Monte Carlo approaches to problem solving are characterized as *"embarrassingly parallel"*. Of course this is true since independent runs can be performed in parallel; however, on the practical side there might be some loss in the expected speedup since we want random sequences with good statistical properties for all processors (cores) that are working in parallel.

Fortunately though, in the problem we are facing, we can still exploit the builtin `C/C++` `rand()` function for the various "random" sequences that are required in independent runs. The reason is that we have an easy upper bound on the calls made to the `rand()` function on each *episode* generated. Typically heroes have all 8 skills at expert by level 23, which implies 22 *level-ups*, and therefore at most $2 \cdot 22 = 44$ calls to the `rand()` function per simulation run. Hence, if the user requests an estimate for the probability distribution of various skills under a

specific strategy as this is evaluated after n episodes, we know for sure that calls to the `rand()` function are no more than $44 \cdot n$.

Hence, the current implementation generates a sequence of length $44n$ with random numbers that can be used later on *worker* threads in the obviously parallel part of the problem. For this purpose a distinct queue is devoted to each of the t threads, and $\lfloor 44n/t \rfloor$ elements are inserted in each queue. The rest (if any) $44n \mod t$ elements are assigned to the thread with the smallest possible id. In order to hide the latency of the generation of the *worker* threads (those that perform the Monte Carlo simulation), the threads are spawned and wait for a signal in order to start working. Some preliminary implementation results indicate that the practical computational cost of an episode (a single simulation run) is computationally equivalent to the generation of about 15 random sequences of length 44 each. Hence, during this first phase of the algorithm we might have to store in memory numbers in the order of $44n$, where n is the requested amount of episodes.

Note that numbers returned by `rand()` lie in the interval $\{0, 1, \ldots, 2^{31} - 1\}$, and therefore a 4-byte integer is required to store each one of them, meaning memory requirements of order $176n$ (in bytes). Typical values for n are a few millions, which in turn implies worst case memory requirements a multiple of 176 megabytes. Moreover, remark 3.1 indicates that we can not *truncate* these numbers and store them in variables of type `unsigned char` (1 byte) for example, by storing the remainder of those values with 256 or in the more intuitive form $\mod 112$ (112 is the sum of all weights of the various skills per hero).

Remark 3.1. Note that $(4 \mod 3) \mod 2 \neq 4 \mod 2$.

In other words, we have no guarrantes that the truncated sequence still preserves good statistical properties, since the partial computation we perform in order to save memory space, in effect alters the properties of the sequence[1]. In a nutshell the phases of the algorithm are:

Phase 1: This phase lasts as long as a *generator* thread[2] produces a random sequence of appropriate size required by all simulation runs by the various *worker* threads. As soon as a subsequence of the appropriate size is generated, the thread that is waiting for that input is signalled to start working.

Phase 2: Once the generator (`main()`) finishes with the production of all the subsequences needed for the various queues, the `main()` waits for any remaining threads to complete their work and report their results.

Phase 3: In this final phase the results are merged and are presented to the user.

3.1 The data structures.

Before I procceed with the actual experimentation I will comment briefly on the data structures that are used to pass data to various threads.

Although, the current implementation does not use a *generator* thread for the entire random sequence which is split to various queues, the data structure is still used in the `main()` function for possible extensions in a future release. The structure has the following form:

[1] This is also verified with the results obtained in practice.

[2] Actually in the current implementation the `main()` function is used for this purpose, so that the overhead of the generation of one additional thread (*the generator*) can be avoided.

```
typedef struct {
   pthread_mutex_t ** queue_mutex_array;
   pthread_cond_t ** queue_cond_array;
   QUEUE_PTR * queue_array;
   bool ** waiting_array;

   int id;

   int items_per_chunk;
} GENERATOR_DATA;
```

queue_mutex_array is an array of pointers to muteces of size WORKER_THREADS; i.e. of size equal to the number of threads that perform the simulation runs. Similarly, queue_cond_array is an array of pointers to condition variables of size again WORKER_THREADS. queue_array is an array of pointers to queues used for storing the necessary subsequences for the various worker threads. Obviously its size is again WORKER_THREADS. waiting_array is an array of pointers to boolean variables again of size WORKER_THREADS. Variable id stores the unique id of each worker thread, and finally items_per_chunk is equal to 44 since this is the length of the smallest sequence needed to be generated so that a single simulation can take place.

Regading the data structure that is used by the worker threads we have:

```
typedef struct {
   pthread_mutex_t * queue_mutex;
   pthread_cond_t * queue_cond;
   QUEUE_PTR queue;
   bool * waiting;

   int id;

   pthread_mutex_t * total_episodes_mutex;
   long unsigned int * total_episodes_atm;
   long unsigned int total_episodes_computed_by_threads;

   long unsigned int episodes_to_compute;

   MONTE_CARLO_PTR myMC;
   MT_PTR myMT;
} WORKER_DATA;
```

The first five variables were explained in the above paragraph. Note that in this case, threads are not passed arrays of pointers, rather than the actual pointers that interest them. The next three variables are used in order to print a message online indicating the percent of the work done at any given time. Variable episodes_to_compute denotes how many runs should be performed by the specific thread. Finally, the last two pointers are used to refer to the MonteCarlo object and its associated mathematical toolbox which respectively perform the simulation and store the results of each run.

3.2 One more note on the implementation

A variation of the above program was also implemented. The structure of this variation was based on a single queue accessed by all the processes *on demand* right from the beginning of the execution. However, it turned out that in practice the overhead required for locking and unlocking a mutex so that each thread can *talk* to the queue resulted in about 5-10% performace loss with respect to the results presented in section 4. Hence this approach was abandoned.

3.3 Where can I get this version?

The program is uploaded in its *natural* location which is [4].

4 Experiments and results.

Experimentation and measuring running times has been more than a challenge. All running times presented below were measured on a MacBook Pro with an Intel Core 2 Duo processor with 2 GB of RAM and 4 MB of L2 Cache. Both the serial program and the parallel one (with threads) were compiled with `gcc` version 4.0.1 :

```
$ g++ -v
Using built-in specs.
Target: i686-apple-darwin9
Configured with: /var/tmp/gcc/gcc-5465~16/src/configure --disable-checking
-enable-werror --prefix=/usr --mandir=/share/man
--enable-languages=c,objc,c++,obj-c++
--program-transform-name=/^[cg][^.-]*$/s/$/-4.0/
--with-gxx-include-dir=/include/c++/4.0.0 --with-slibdir=/usr/lib
--build=i686-apple-darwin9 --with-arch=apple --with-tune=generic
--host=i686-apple-darwin9 --target=i686-apple-darwin9
Thread model: posix
gcc version 4.0.1 (Apple Inc. build 5465)
$
```

However, it turns out that `apple` architecture above produces a very smart version of the serial implementation which perhaps exploits partially both cores while executing; the user can actually verify on the `Activity Monitor` that the program changes cores while executing. In the respective `makefile`s that were used in both cases, the requested machine architecture was the deprecated `x86-64`. Both programs were compiled at optimization level 3.

Table 1 presents running times for some sample heroes on the AR and AL policies (strategies) for $5,000,000$ episodes (runs). The heroes were chosen according to the state-space that they imply in various cases. Thane, Crag Hack, and Rashka are all heroes starting with only 1 skill. What differentiates them is that Thane can acquire 27 out of the 28 possible skills, Crag Hack can acquire 26 out of the 28 possible skills, while Rashka is a class on it's own in the sense that he is the unique *mighty* hero that starts with Wisdom, which forms an exception group on its own. Apart from them, Orrin and Ivor are both heroes starting with two skills where Orrin can acquire 27 out of the 28 possible skills, while Ivor can acquire 26 out of the 28 possible skills. These heroes are representatives for equivalence classes (w.r.t. the induced state-space that has

to be explored) that span all the *mighty* heroes. Note that *magic* heroes are not supported even on the serial implementation. In figure 1 one can view the speedup achieved in each case.

Hero	Serial		Threads	
	AR	AL	AR	AL
Thane	65.76	71.00	37.34	38.96
Crag Hack	64.69	71.70	37.29	41.95
Rashka	65.53	71.86	37.09	38.71
Orrin	63.87	70.92	36.87	38.88
Ivor	63.89	71.24	37.04	38.83

Table 1: Running times (secs) for simulating 5 million episodes.

Hero	Speedup	
	AR	AL
Thane	1.76	1.82
Crag Hack	1.73	1.71
Rashka	1.77	1.86
Orrin	1.73	1.82
Ivor	1.72	1.83

Figure 1: Speedup achieved.

Similarly, figure 2 records the efficiency achieved in each case. Apart from the policies shown above, ALTP and SPOU are also supported. The reason that they are not included in the above tables boils down to the fact that both of these strategies expect additional input by the user as to his selection on skills. Hence, it was difficult to generate a file that can be used for this purpose and simultaneously cover all of the above heroes, as well as have some meaning with the actual preferences of the users' on various skills. Nevertheless, some preliminary results indicate that in these more complex and interesting skill-selection strategies the speedup as well as the efficiency achieved is boosted, w.r.t. the values above. As an example, Damacon and Gunnar on an identical preferences file have achieved a speedup of 1.93 or in terms of efficiency 0.97. The terminal output of the above experiments is available online in [4]. Refer to the *Home Log* near the end of the homepage to retrieve the output.

5 Future work.

Hero	Efficiency	
	AR	AL
Thane	0.88	0.91
Crag Hack	0.87	0.86
Rashka	0.89	0.93
Orrin	0.87	0.91
Ivor	0.86	0.92

Figure 2: Efficiency achieved.

The first and foremost concern of practical importance is to port the above approach under Windows, since this is the native platform of the game and the area where the main audience lies. [6] seems to be the solution in this direction; yet needs to be tested.

Another thing that might need further consideration was mentioned in section 3. At the moment, the algortihm spends its first phase with a single thread producing one big random sequence, that is decomposed in smaller subsequences which are lead to the queues serving the threads that actually perform the computational task. Another approach would be to use a library that allows distinct random number generators per thread; e.g. [8, 7]. This would eliminate some overhead, and parallel execution would be much more efficient regardless of the decisions of the scheduler. Moreover, each dedicated random number generator would produce numbers on a *per-request* basis, and as a result

the entire process would now be actually *embarrassingly parallel*. But the most important practical contribution of this approach would be the fact that the enormous (practical) memory requirements would be diminished. On the other hand, the efficiency of those generators with respect to the `rand()` function remains to be tested in practice and verify that they are a viable solution for this purpose.

Other approaches might also be considered in such a way as to eliminate the possibility that the scheduler delays our *generator* (of random numbers) thread. In this direction it might be useful the fact that we can get the number of cores of a specific machine with the command `sysconf (_SC_NPROCESSORS_CONF)` of the `unistd.h` library.

References

[1] Michael Allen and Barry Wilkinson. *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2 edition, 2005.

[2] dimis. On the internals of offered skills when leveling-up a hero. *Library of Enlightenment*, April 2006. http://heroescommunity.com/viewthread.php3?TID=17812.

[3] dimis. The 'Always New Skill Advancement' (ANSA) problem. *Library of Enlightenment*, April 2006. Post 3 at http://heroescommunity.com/viewthread.php3?TID=17812&pagenumber=1.

[4] dimis. internals_mc: Evaluation of user's policy with Monte Carlo methods. *Library of Enlightenment*, July 2007. Post 10 at http://heroescommunity.com/viewthread.php3?TID=17812&pagenumber=8.

[5] dimis. On the offered skills in Heroes of Might and Magic III. *Library of Enlightenment*, January 2008. Post 4 at http://heroescommunity.com/viewthread.php3?TID=24181&pagenumber=1.

[6] Ross Johnson. Open Source POSIX Threads for Win32. GNU Lesser General Public License. http://sourceware.org/pthreads-win32/.

[7] Michael Mascagni, Hongmei Chi, and Jane Ren. The Scalable Parallel Random Number Generators Library (SPRNG). Florida State University. http://sprng.cs.fsu.edu/.

[8] Steven Skiena. Parallel Random Number Generation. Stony Brook University. http://www.cs.sunysb.edu/ algorith/implement/myftp/implement.shtml.

[9] Xarfax111. The Library's Heroes 3 Tribute Thread. *Library of Enlightenment*, November 2007. http://heroescommunity.com/viewthread.php3?TID=24181.