# Some remarks on the execution speed using multi-processor systems

Orsolya Gereben 2010-06-29

The following discussion is based on the speed tests made for RMC_POT. The program were tested first on the following machines:

- Computer (A) 2 cores: AMD Athlon(tm) 64 X2 Dual Core Processor 4600+ models (64-Kbyte 2-Way Associative ECC-Protected L1 Data Cache/core with Two 64-bit operations per cycle, 3-cycle latency; 64-Kbyte 2-Way Associative Parity-Protected L1 Instruction Cache/core with advanced branch prediction; 512 Kbytes 16-Way Associative ECC-Protected L2 Cache/core; Core Speed 2400 MHz, System bus speed 2000 MHz). GNU Linux reports cache alignment 64, so cache line size is supposed to be 64 Byte, but no definite information was found about this.
- Computer (B) 2 cores: Pentium(R) Dual-Core CPU E5200 @ 2.50GHz, Core speed 2500 MHz, Bus speed 800 MHz, no L1 instruction or data cache, 2 MB L2-cache
- Computer (C) 8 cores: Intel(R) Xeon(R) CPU E5345 2.33GHz computer containing 2 quad-core E5345 processors, so having the total number of 8 cores. One quad core processor is one physical package containing inside 2 dies, 2 cores/die with 8-way associative 32 KB L1 data and 32 KB L1 instruction cache/core and 16-way associative 4MB L2 cache/die.
-

The speed of the programs was tested the following way: the simulation was always started from the same initial configurations using the same parameters, and _TEST_MODE command line option was switched on for the compilation (the random number generator started with the same seed, and the number of generated steps were the same) so the final configuration was the same as well. The total running time in the loop reported in the **.hst* file was compared, the efficiency was calculated according to the formula:

$E(p) = \frac{C_1}{pC_p} 100\%$, where $C_1$ is the running time of the program using consecutive execution, $C_p$ is the running time of using *p* threads, where each thread is executed by its on processor core. The basic structure of RMC for multi-threading is that it has a main thread which creates the auxiliary threads not far from the beginning of the program and gives task to them, where it is possible. The main thread handles the tasks, like the parameter reading and smaller calculations, which could not be or was not worth splitting among the threads. The farm algorithm was used for splitting the work among the threads for the computationally heavy parts of the program, and the main thread always takes equal share in these calculations as well. To avoid excessive mutex usage, the size of some arrays (as the histogram ptotal and pcounts and the coordnumbs and nsatisfy arrays) were multiplied by the number of threads, so each thread had its own segment to write parallel, and when each thread finished with the calculation of its own segment, the results is combined and the main thread's segment (representing the complete array) is updated.

## Speed loss due to false sharing

In the case of the 8-processor system, quite substantial speed loss was experienced using higher number of threads >4, if the segments of the different threads followed consecutively each other due to 'false sharing'. False sharing happens, when the different threads are trying to write different memory locations, but these location's addresses are close to each other and they are cached together forming the same cache line. Lets assume as an example, that

ThreadA (executed by CoreA) wants to update the array elements [0-99], ThreadB (CoreB) [100-199], ThreadC (CoreC) [200-299] of the same array referred to as Tarray. The cache strategy of different processors can be different, but for the used Quad-core Intel(R) Xeon(R) CPU E5345 processor the cache line is 64 bytes for all caches. Always a full cache line is cached at a time, regardless the size of the given variable to be modified, and the processor always starts a cache line beginning on a 64-byte boundary (64 aligned cache, where the beginning address' 6 least significant digits are 0), so the desired memory location might be in the middle of the cache line. That means, that variables having the neighboring addresses will be cached together with the desired variable. After modifying the desired variable, the whole cache line, and not just the modified part of it is flushed (written back) to the memory, so that part of the memory is locked for the time of the writing. Lets assume, that Tarray is an integer array with the usual integer size of 4 bytes, so 24 integer elements are cached together. Lets also assume, that the beginning of the array, Tarray[0] starts at a 64-byte boundary. When for example Tarray[97] is changed in the L1 cache of CoreA and simultaneously CoreB in its own L1 cache modifies Tarray[105] than both of them wants to write back the same whole cache line (containing Tarray[96]-[119]) to the same place in the memory, and they are holding up each other. The actual picture is even more complicated, because if this would happen simply, than the core (lets say Core B) updating the memory after the first core (CoreA) finished with its own update would write back the original, unmodified value of Tarray[97] overwriting the already updated value. This will not happen, because snooping occurs, meaning, that if the same cache line can be found in different core's cache, then if one modifies one part of the cache line, then the other will update its own cache with the modified data coming from the other core before further updating it itself or the memory. It has to be emphasized, that snooping provide cache consistency, if different parts of the same cache line are modified by different processors, but the programmer has to make sure, that different threads of the program will not try to update the same variable in the same time, because in this case the result is unpredictable!

Obviously false sharing will only become a problem, if it happens very often during program execution.

False sharing can be avoided by cache padding, which means that dummy elements has to be introduced between the array segments of the different threads to prevent the elements of different threads being cached in the same cache line. As we do not know beforehand, where the cache alignment boundaries will be situated inside the array, we have to make sure, that even in the most unfortunate case there are enough dummy elements to separate the elements of the different threads. The following formula will safely give the number of dummy elements to separate the thread segments for Tarray:

(CACHE_LINE_SIZE – size_of(*Tarray))/size_of(*Tarray), where size_of (*Tarray) is the number of bytes occupied by one element of the array.

It has to be noted, that on some platforms not only one cache line is cached for a desired memory location, but due to pre-fetching policy a sector (for example two consecutive cache line). It also depending on the platform, whether a cache line or the whole sector is flushed back to the memory if one variable was modified. The CACHE_LINE_SIZE has to be the size in bytes, which is flushed back to the memory after one variable, is modified.

Cache padding was used in RMC_POT for HistoSet::pcounts and ptotal arrays and in CoordNumConst::nsatisfy, and the large speed difference disappeared among the threads for the multi-threaded program parts, and the performance improved.

## Length of some integer variables

In some cases, if the system size and/or the number of steps is large, it can be necessary to use 8-byte integers for certain variables. The size of the integer variables declared as **int** is usually 4 byte, but the size of the **long int** varied for the different computers, it was 4 byte on a Intel(R) Pentium(R) 4 CPU 3.00GHz hyper-threading computer using Windows and Microsoft Visual C++, but it was 8 byte on the computers A-C used for the speed tests having Linux operation system and using Debian gcc 4.3.2 compiler. Therefore those integer variables, where it can be necessary to change their size easily were declared with the custom type longint, and its actual size depends on the compilation. If _USE_INT64 compiler option is switched on, (I64=0) is passed to Linux make).

## Speed test results

### Normal RMC run

The test system was a rather large configuration containing 114240 atoms of one type with $\rho$=1.2148·10-$^{6}$ number density, and used an *S(Q)* data set with 190 points and a coordination constraint with 13 sub-constrains.

The 2-threaded efficiency was **88 %** on the 2-processor system (B) regardless int or long int was used for typedef longint.

The 2-threaded efficiency was **81 %** on the 2-processor system (B) regardless int or long int was used for typedef longint.

The result for the 8-porcessor computer system (C) was ~98 % for the 2-threaded and 77-81 for the 8-threaded efficiency, see Table 1 for details.

| System B | conse-cutive | parallel | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Nthreads | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| t (s) longint=int | 1146 | 1143 | 580 | 401 | 308 | 259 | 226 | 196 | 177 |
| E(p) (%) | | 100 | 98.6 | 95.0 | 92.8 | 88.3 | 84.3 | 83.3 | 80.7 |
| t (s) longint=long int | 1142 | 1143 | 581 | 402 | 314 | 264 | 230 | 204 | 186 |
| E(p) (%) | | 100 | 98.4 | 94.8 | 91.0 | 86.6 | 82.8 | 80.0 | 76.8 |

**Table 1: The speed test for the 8-processor computer using integer or long integer for the typedef longint variables.**

### Local invariance

The speed-up with the usage of local invariance calculation was tested on System (C). The test system consisted of 4000 atoms and had an S(Q) data set with 228 points a coordination constraint with 13 sub-constraint and an average coordination constraint and the local invariance was calculated as well, see Table 2.

| System B | parallel | | | |
|---|---|---|---|---|
| Nthreads | 1 | 2 | 4 | 8 |
| t (s) longint=int | 1873 | 883 | 441 | 240 |
| E(p) (%) | 100 | 106 | 106 | 98 |
| t (s) longint=long int | 1875 | 877 | 442 | 106 |
| E(p) (%) | 100 | 107 | 106 | 97 |

**Table 2: The speed test for a local invariance calculation for the 8-processor computer using integer or long integer for the typedef longint variables.**

**Non-bonded potential**

The speed-up with non-bonded potential calculation was tested on System (C) for the same large (114244 atoms) system as for normal RMC simulation test with the same data set and constraints using neutral particles with 30 A for the vdW cutoff.

| System B | conse-cutive | parallel | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Nthreads | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| t (s) longint=int | **1184** | **1213** | **597** | **410** | **320** | **268** | **233** | **204** | **184** |
| E(p) (%) | | **100** | **101.6** | **98.6** | **94.8** | **90.5** | **86.8** | **84.9** | **82.4** |
| t (s) longint=long int | **1230** | **1188** | **601** | **414** | **323** | **274** | **238** | **211** | **189** |
| E(p) (%) | | **100** | **98.8** | **95.7** | **92.0** | **86.7** | **83.2** | **80.4** | **78.6** |

**Table 3: The speed test for a non-bonded potential using simulation for the 8-processor computer using integer or long integer for the typedef longint variables.**

**Summary**

Large speed up can be expected with larger system sizes (where the histogram calculation would amount for a considerable time of the total running time, lots of coordination constraint (sub-constraints), which is parallelised as well, or more large data sets, because the Fourie-transformation is parallelised as well, so it is very much depending on the applied simulation system. As the speed test showed, even for the same test system and same code the speed-up can strongly depend on the computer architecture as well, so writing a code, that would perform equally well on different platforms might not even be possible. It is worse trying to compile the code and run some speed test with increasing the NUMBER_OF_CACHE_LINES_TO_FETCH parameter in the *units.h*, especially if you are not sure in your architecture specifications. The difference can be substantial usually in case of using >4 processors.