STUDIA INFORMATICA 2014
Volume 35 Number 4 (118)

Krzysztof A. CYRAN, Dariusz MYSZOR
Silesian University of Technology, Institute of Informatics

MULTITHREAD PARALLELIZATION OF A RAPID COALESCENT-BASED WHOLE GENOME SIMULATOR

Summary. In this article improvements introduced into GENOME: A rapid coalescent-based whole genome simulator were presented. Implemented modifications allowed for obtainment of significant speed up of the process of results obtainmenvol.

Keywords: coalescence, multithread

WIELOWĄTKOWE ZRÓWNOLEGLENIE SZYBKIEGO SYMULATORA CAŁEGO GENOMU OPARTEGO NA KOALESCENCJI

Streszczenie. W artykule przedstawiono modyfikacje wprowadzone do kodu aplikacji GENOME: A rapid coalescent-based whole genome simulator. Wprowadzone zmiany mają na celu skrócenie czasu oczekiwania na wyniki dzięki wykorzystaniu wielowątkowości.

Słowa kluczowe: koalescencja, wielowątkowość

1. Introduction

Ongoing advances in biological researches allow for sampling and analysis of huge amount of data obtained from existing populations [1, 2, 3], despite of this fact computer and mathematical models are still important tools in many biological fields [4, 5]. It happens because application of these methods usually allows for obtaining the results in a much shorter timeframe than execution of regular experiments. At the same time utilization of computer simulations often brings additional benefits in the area of cost reduction. Computer simulations, although often simplify analysed processes, are powerful tools. In some cases results obtained through computer simulations can lead to counterintuitive conclusions, that are later

proved by experimental verification. In addition, with every year computational power of machines is increasing, and at the same time cost of computing hardware (in the relation to its computational power) is decreasing. It is worth to mention that recent trends of cloud computing allow for easiness on demand of computational power obtainment [6, 7, 8], such abilities can be useful when there is a need of conduction of demanding simulations, however shortage in computational equipment occurs.

Computer simulations are widely utilized in the field of population genetics, in order to create artificial sets of data that can be utilized for validation of various hypothesis and analysis methods. Main purpose of population genetics is analysis of distribution and changes of frequencies of alleles in the population [9]. Therefore simulation methods are often focused on efficient creation of set of individuals with chromosomal representation. Interestingly, simulated chromosomes might have various representations, they can consists of individual nucleotides, genetic markers (e.g. microsatellites) [10] or whole genes.

Most popular methods of population development simulation include time-forward and time-backward approaches. In time–forward approach [11], creation of all successive generations is conducted. During the process of simulation of every generation all individuals, from effective population size, are created. Then for each newly created individual parents, from the previous generation, are assigned. At the same time, decision is made about occurrence of various processes such as mutations and recombination. This approach allows for huge flexibility in terms of population structure, and introduction of various processes as well as interactions. However, creation of every individual has to be simulated and therefore, there is a huge downside in the time of generation of successive generations. In addition, usually a lot of computational power is wasted, because genetic drift leads to the elimination of significant fraction of first generations descendants so, unless some interaction between individuals are taken into account, these individuals do not have influence on the structure of the final generation. Time – forward methods also require proper initialization of first generation of individuals or application of warm up period, during which simulated data should lost traces of initialization with random data.

Time-backward approach [12, 13, 14], on the other hand, tracks individuals backward in time, from the final population to the generation which possesses most recent common ancestor (MRCA). It allows for omission of lineages that extinct in the process of population developmenvol. In addition utilization of coalescent theory [15, 16], through incorporation of mathematical apparatus into simulation software, allows for simulation of generations in which coalescent event occurred i.e. two (or more) individuals possessed common ancestor. Development of coalescent theory allowed for efficient simulation of many phenomenon such as recombination, natural selection etc. However scientists noted that when analysed sequences are long then coalescent events occur in almost every generation. Therefore, in-

stead of constant calculations of coalescence time and omission of analysis of uninformative generations, more efficient solution relies on tracking of ancestors of individuals in every generation. It allows for significant reduction of the amount of calculations which have to be conducted, thus reduction in result await time can be achieved.

In this article improvements introduced into GENOME: A rapid coalescent-based whole genome simulator, which is based on coalescent methodology, are presented.

2. Base solution

As a base for performance improvements, GENOME: A rapid coalescent-based whole genome simulator [17] has been chosen. It was created by Center for Statistical Genetics, University of Michigan. It is single thread application written in C++. Executable files and source code are available at authors' web page and allowance for redistribution with modification is given, if specific conditions are mevol. Conditions are as follows:

1. Redistributions of source code must retain specific copyright notice, this list of conditions and specific disclaimer.
2. Redistributions in binary form must reproduce the specific copyright notice, this list of conditions and specific disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of project contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

Huge advantage of the solution is its portability, code can be compiled on variety of operating systems without the need of introductions of any modifications. It was achieved through limitation of utilization of external libraries. In order to achieve further independence from third party libraries source code of Mersenne Twister pseudorandom values generator was incorporated [18].

Execution of application is divided in two phases. The first one allows to trace back in time alleles of genes of individuals, which were sampled from the final population, up to the point when most recent common ancestor for each gene type is available. During this phase processes of coalescence, recombination and inter-subpopulation migration are simulated. Interestingly when long genomes are considered coalescent events occur at almost every generation, therefore in order to simplify algorithm, and as a result to shorten gene genealogy obtainment time, every generation is simulated. During the second phase of application execution, mutations are applied on obtained gene genealogy.

In order to run the application following input values are required: number of subpopulations (p) effective subpopulation size (N), number of individuals sampled from every sub

population (n), number of independent regions (chromosomes, c), number of fragments per independent region (genes, g), length (number of nucleotides, l) of each fragment, recombination rate (r), migration rate (m), mutation rate per generation per base pair (mt), fixed number of SNP per chromosome (s).

Original application was written in C++ and was based on a single process which spans a single thread. In order to decrease result await time, decision was made that multithreading will be introduced.

2.1. Randomness

Coalescence events require stream of values which captures all important statistical properties of true random sets of data. Creators of a single thread version of the code utilized Mersenne Twister pseudorandom value generator which is widely applied because it allows for efficient generation of stream of consecutive values, ensures low memory utilization and at the same time is characterized by huge period equal to 2^{19937}-1 .

There are two ways of utilization of PRNG in multithread applications. The first one assumes that there is only one available PRNG object which is shared among all threads, the second introduces individual PRNG objects for each thread. For single PRNG approach, generator is initialized once, at the beginning of the simulation process. Then every thread queries PRNG for consecutive pseudorandom values. Such an approach allows for significant simplification of the code in the area of PRNG initialization, however it also complicates implementation of PRNG, because semaphores must be utilized in order to prevent simultaneous access to PRNG code by many threads. MT PRNG is a good candidate for implementation of such an approach because it operates in two phases which can be called data obtainment phase and data generation phase. Data generation phase is responsible for generation of an array of 624 consecutive pseudorandom values, and initialization of a pointer which is responsible for tracing of next unused value from an array. In the data obtainment phase, on the other hand, values are taken from an array and pointer is shifted to the consecutive value. Then obtained value is being modified with tempering transform. If there are no unused values in an array, data generation phase is executed. Critical section must enclose data generation phase and part of data obtainment phase related to the extraction of data from an array and shifting of a pointer. Importantly in multithread environment data tempering phase operates on independent data therefore it can be executed concurrently by many threads.

When each thread possesses its own PRNG object, there is no need of critical section application, however necessity of initialization of seed for every PRNG occurs. Naive approach might include generation of seed by every thread, however then there is a possibility of oc-

currence of duplicated seed values among threads. In order to rule out such a possibility one thread should be responsible for generation, validation and distribution of PRNG initialization values. This approach of seed generation was applied in suggested solution, main node generates consecutive seeds based on additional random number generator, which was seeded with time.

Both approaches were implemented, in order to analyse performance impact and allow for determination of the best approach. Results point out that utilization of a single PRNG for single thread has negative influence on the performance, therefore many independent PRNG objects were utilized in final version of an application.

2.2. Multithreading

Proposed solution assumes that application will be run on a single computer which possess many processor cores. Huge advantage of contemporary computers is that they can posses many processors which are equipped in many computation cores, in addition each core can be designed in order to process many threads at the same time. As a result computation server equipped with 2 processor, 6 cores each, that are able to run 24 threads (e.g. configuration based on Intel Xenon processors) at the same time are widely available. Utilization of current processors in combination with proper amount of memory, that is shared among computational units, allows for fast exchange of data between threads and facilitates thread synchronization. If single thread application can be rewritten in a way which takes benefits from multithread abilities then significant performance gain can be achieved.

2.3. Data structures

In the final generation each sampled individual has full set of genes assigned, then simulation goes back in time. As a result of recombination process, child genome might be composed of genes derived from more than one parent, therefore only part of parental genome might be inherited in the next generation (see fig. 1). In order to speed up the process of simulation and reduce memory consumption, at the level of parent only genes which influence individuals in the next generation are created (so called influential genes). Therefore creators of the original code based the solution on set of sparse arrays, which are responsible for representation of individuals and their influential genes.

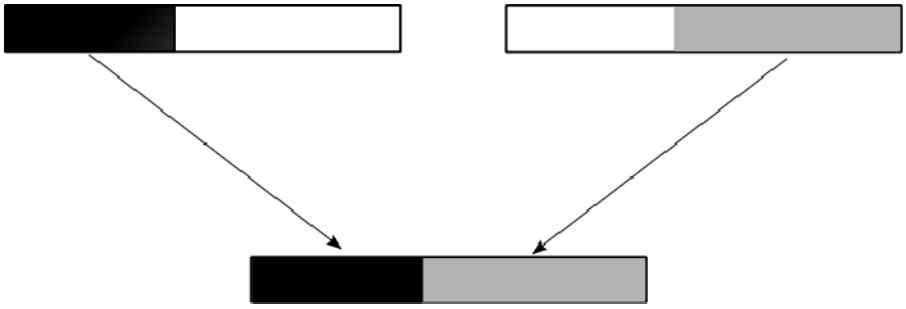

Fig. 1. Recombination process
Rys. 1. Zjawisko rekombinacji

At the same bottom continue block of memory, called third level array, is located. Gene pool of each individual is divided into blocks, 128 genes each, only blocks which possess at least one influential gene have representation in the memory. When such block is created, free area (128 cells) of third level array is assigned and all locations are initialized with -1. Later at locations, which represent influential genes, identification number of these genes are stored. Importantly the process of assignation of free location in third level array is based on free memory pointer variable, that is holding index of next free block of data in third level array. This counter is updated appropriately each time new block is assigned.

Pointers to appropriate locations in third level array are hold by set of second level arrays. Each individual possesses one second level array. If given block does not possess any influential genes, then it does not have representation in the memory, is such a case respective cell in second level array has NULL assigned. At the top is first level array which represents index of each individual from the population (it has N*p cells). In this array pointers to second level arrays are located. If individual is not sampled from the population (for final generation) or it does not have any descendants (in consecutive generation) then NULL is assigned to respective cell in first level array.

Two sets of such structures are created, one for currently analysed generation and one for previous generation. At the end of the process of analysis of every generation, appropriate pointers are switched, so variables which point to the current generation structures are moved to the previous generation structures, and variables which point to the previous generation structures are moved to the current generation structures. As a result there is no need of allocation of new structures when next generation is simulated, because parental structures (previous generation) becomes descendant structures (current generation) and structures which represented descendants, after the switch are reinitialized and ready to represent parents. In addition there is a set of arrays which are holding gene genealogy and coalescence times in order to be able to append mutations during execution of the second phase of the simulation process.

In proposed multithread solution all threads share above mentioned data structures, as a result communication is simplified and significant speed gain might be acquired, however

introduction of shared memory approach results in necessity of application of critical sections in areas of code where shared resources are utilized. Three are 4 mutex applied in order to protect the memory against concurrent operations issues. Critical section must be introduced at the level of accessing and updating of free memory pointer variable, during the process of assignation of memory in third level array for newly created block. Array of mutex is created (size equal to effective population size) in order to protect structures assigned to single parent against concurrent access by many threads. Mutex must be applied at the level of modifications of additional arrays which store gene genealogy and gene coalescence times

Third level arrays have limited sizes, in some cases size might not be sufficient in order to store all the data (especially when recombination is high), in such a case reallocation is required, however many threads have access to this structure at the same time, so immediate reallocation is not an option. Therefore threads, which are hitting this limitation, set the flag which informs that arrays should be reallocated and current generation should be simulated again.

As mentioned earlier execution of simulation application is divided into two phases. First one is responsible for gene genealogy creation, second one is responsible for application of mutation process. Preliminary researches point out that first phase takes significant fraction of time of the whole simulation (see tab. 1), therefore focus was put on speed up of processes ongoing during the first stage.

Table 1

Performance analysis of a single thread solution

Simulation description	Phase 1 [s]	Phase 2 [s]	Phase 1 [%]	Phase 2 [%]
N=10000 n=6000	17.9194	70.09069	0.203606	0.796394
N=100000 n=6000	6.931286	424.5023	0.016066	0.983934
N=100000 N=60000	181.9955	493.5936	0.269388	0.730612

Single thread application, during the first phase of operation, scans through every cell of first level array and if individual is located search, for its predecessors in the previous generation, is executed. This process is repeated until every most recent common ancestor of all genes types is obtained.

In order to be able to create multithread application, Thread class from Boost library was utilized. It is widely known free library which can be utilized on various operating systems, thus portability of final product is obtained. Multithreading at many levels can be introduced, however deeper analysis suggests that the best solution is to introduce concurrency at the level of predecessors identification process.

The first multithread implementation was based on the concept that for the purpose of analysis of every individual (in every generation) new thread will be created and executed. It allowed for easy management of sparse data structures after processing of every generation. However process of spanning threads took too much time, therefore performance of an application was significantly worse, than base solution. Boost allows for utilization of pool of threads, which automatically restricts the number of threads and reuses them. When this component was utilized some gain was achieved, however obtained simulation times were still worse than base solution times.

Then conception was changed, effective population size was divided into chunks of individuals, each thread was responsible for processing of a single chunk, what is more thread were spanned only at the beginning of simulation and were operating until simulation was finished. It introduces an issue with sparse structures management, after simulation of every generation. Decision was made that thread with id equal to 0 (so called master thread) will be responsible for this task, as a result necessity of thread synchronization was introduced. Two synchronization points, during simulation of a single generation, are required. During analysis of a single generation, all threads have to wait on the first barrier until all threads process all individuals from the individuals' pool. In the next step master thread should manage data structures in order to prepare them to simulation of the next generation. Rest of the threads should wait on the second synchronization barrier, in order not to touch structures which are modified by the master thread. When process of management is completed all threads can proceed to simulation of the next generation. This solution rules out the main issue of the first approach, because threads are spanned at the beginning of the simulation process. However synchronization barrier has to be introduced. Initial approach assumes utilization of Barrier class from Boost library, however performance issues resulted in implementation of custom synchronization barrier, described in the next section.

2.4. Custom barrier implementation

Boost library offers synchronization primitive which can be utilized in order to cause a set of threads to wait until all threads from the set achieve certain point in their execution. Class is available under the name Barrier and is directly available in Boost namespace. Application of barrier provided by Boost is relatively simple, barrier constructor takes the number of threads that should be synchronized, every eligible thread executes wait method in the point of synchronization and then is blocked. When defined number of threads reach synchronization point, blocked threads are placed in ready state. As a result, they are able to carry on their operations. Unfortunately preliminary performance analysis point out that application of this primitive leads to significant extension of result await time. For example

when two threads were applied, first phase of simulation process took 3.5 seconds, within this barrier processing took staggering 3 seconds. Situation get worse when number of thread was increased. The decision was made that custom mechanism, which will play role of thread barrier, would be implemented. Proposed synchronization mechanism consists of unsigned int variable (called barrierBitArray) which is treated as bit array. It implies that at least 16 cells are available, each cell corresponds to single thread, therefore 16 threads can be served. If more threads should be applied there is an easy way to move this limit further by application of a different data types such as unsigned long (32 threads) or unsigned long long (64 threads).

Thread is able to pass through synchronization barrier only if corresponding bit in bit array is set to 0. Bit array is initialized with 0 for every location, for which corresponding thread exists, and with 1 for all other locations. When thread reaches synchronization barrier it sets corresponding bit to 1, then it constantly checks in a loop whether corresponding bit is set to 0. When all threads reach synchronization point then all values within bit array are equal to one (thus variable is equal to UINT_MAX), it allows for easy and efficient determination of condition when lock should be released, by simple comparison of two values. In such a case one of the thread, selected at the beginning of simulation, is responsible for initialization of bit array with zeroes at bits linked with threads, as a result threads are able to pass through synchronization barrier.

Additional mutex is required in order to prevent modifications of barrierBitArray by many threads at the same time. In order to prevent application of compiler's optimizations of barrierBitArray, which might cause that threads does not refresh correctly value of this variable, and as a result lock permanently at on the barrier, volatile keyword should be applied.

2.5. Test environment

All tests were performed on computer equipped with

- Intel ® Core ™ i7-3720 QM processor, that possess 4 cores, HyperThread technology was enabled therefore 8 threads were available. TurboBoost 2.0 technology was turned off and HighPerformance profile was activated thus all cores run constantly at 2600 MHz.
- 16 GB or DDR3 (PC3-12800) RAM clocked at 800 MHz.
- Windows 7 SP1 Enterprise x64 operating system.

In order to rule out influence on 3rd part applications, system was restarted and all unneeded components were turned off before processing with determination of the time required in order to perform tests.

3. Results

In order to compare performance of base solution and implemented multi thread application an array of test was performed. Default values, of input variables were as follow: p=1, N = 10000, n = 6000, g = 30, r = 0.0001, m= 0.00025, l=10000. For every test scenario 10 independent simulation runs were performed.

For default values of input parameter significant speed gain is observed for multithread solution (fig. 2). In general when number of computation threads is increasing, results await time is decreasing up to the point when 4 threads are created. Further increase in the number of threads results in degradation of simulation time. It might be caused by two issues, first is the architecture of CPU which has 4 computation cores, each able to execute two threads at the same time. When core must server two threads, speed of single thread processing might extend. The other issue might be conflicts in the memory access, critical sections must be utilized in order to protect data against concurrent write operations. It can cause emergence of bottlenecks around the areas of code which are often executed and must be protected. In order to validate above mentioned hypothesis, additional machine was utilized. It was equipped with 2 Intel® Xenon® E5 – 2620 (clocked at 2.1 GHz) processor and 128 GB of RAM. Each possess 6 cores, every core is able to run two threads at the same time. Results, presented on fig. 2 seems to confirm assumption that memory access is limiting factor on the way to further utilization of computational power. Nevertheless success was achieved because even 40% of processing time could be cut in some scenarios.

In the next step influence of the effective population size was analysed, therefore two versions of the simulation application were tested (multithread version utilized 2, 4 and 6 threads) for N in the range between 1000 and 100 000 of individuals. Results clearly show that with N time of computation is rising (fig. 3). It is obvious because more individuals in effective population significantly reduces probability that two individuals share common ancestor in the previous generation. Still presented results point out that multithread version is faster in all scenarios than single thread approach, in addition 4 threads seems to provide the best performance for vide range of N.

When size of sampled sample was being modified (n in a range between 6 and 60000), achieved results, presented on fig. 4 indicates that there is minor extension of simulation time. Similar situation is visible for migration coefficient, which modification seems not to have influence on the time of execution of simulation (see fig. 5). On the other hand, results presented of fig. 6 point out that recombination coefficient seems to extend the time of execution of application.

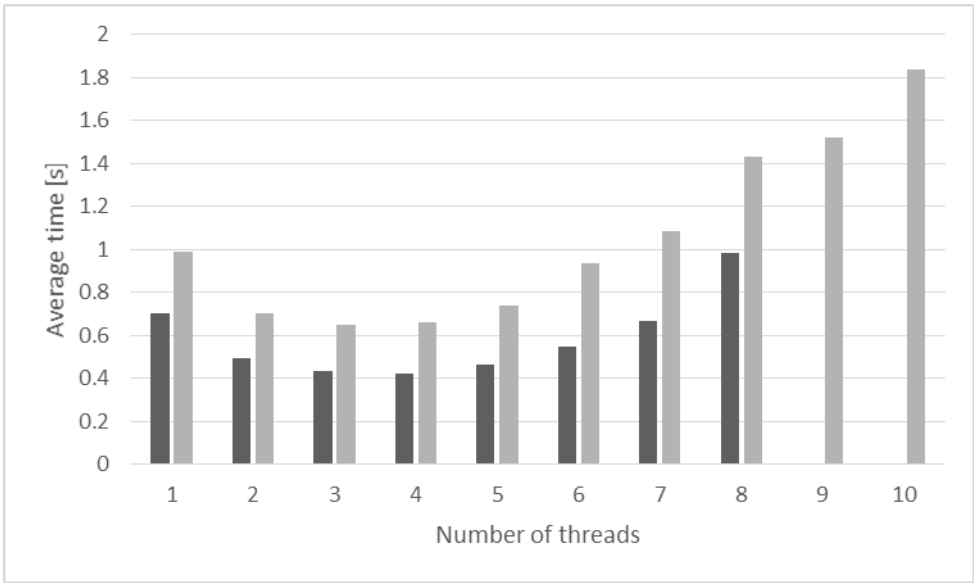

Fig. 2. Time of simulation application execution for various number of involved threads (p=1, N = 10000, n = 6000, g = 30, r = 0.0001, m= 0.00025). Intel® Core™ (dark gray), Intel® Xenon® E5 (light gray)

Rys. 2. Czas wykonywania programu symulacyjnego dla różnej liczby wątków (p=1, N = 10000, n = 6000, g = 30, r = 0.0001, m= 0.00025). Intel® Core™ (ciemnoszary), Intel® Xenon® E5 (jasnoszary)

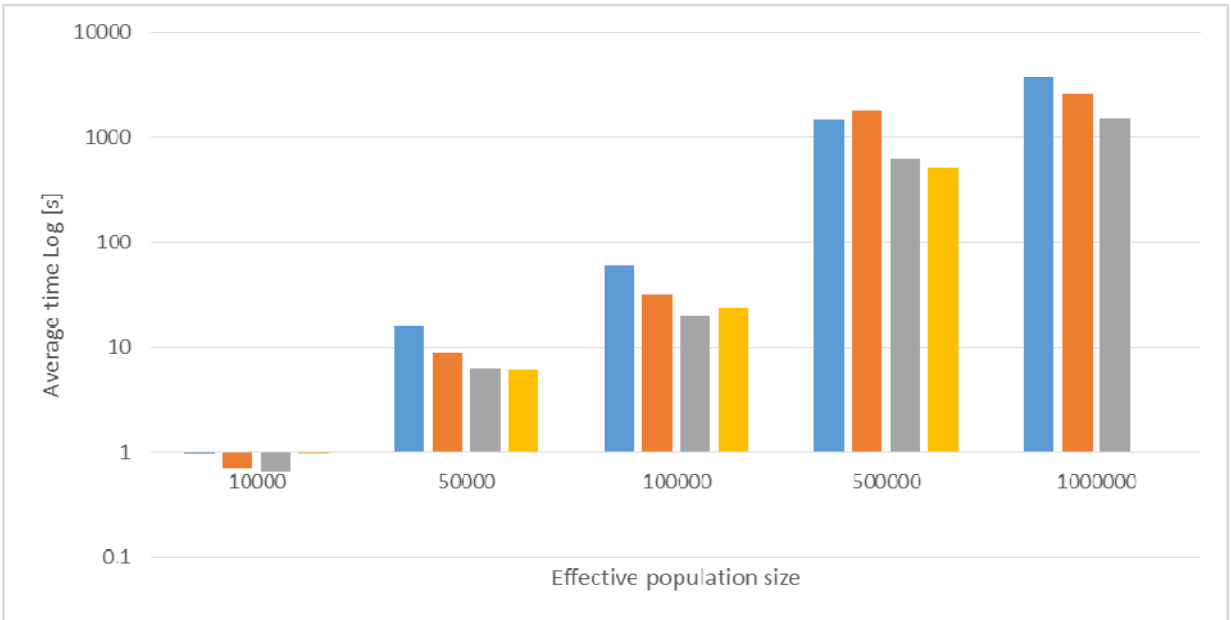

Fig. 3. Time of simulation application execution for various effective population size (p=1, n = 6000, g = 30, r = 0.0001, m= 0.00025). Values on Y axis are in logarithmic scale. Number of threads 1 (blue), 2 (red), 4 (silver), 6 (yellow)

Rys. 3. Czas wykonywania programu symulacyjnego dla różnego efektywnego rozmiaru populacji (p=1, n = 6000, g = 30, r = 0.0001, m= 0.00025). Wartości na osi Y są w skali logarytmicznej. Liczba wątków 1 (niebieski), 2 (czerwony), 4 (szary), 6 (żółty)

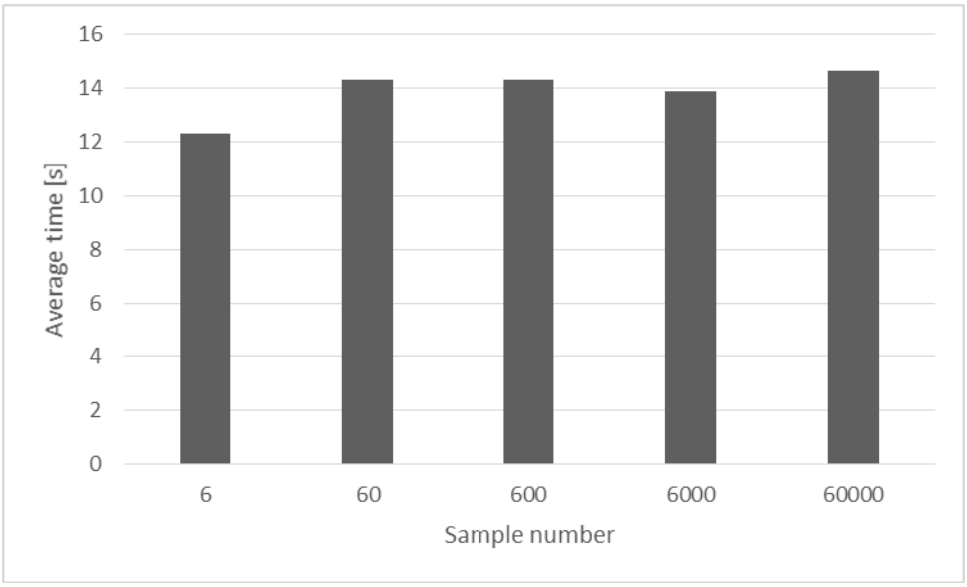

Fig. 4. Time of simulation application execution for various number of individuals in sample taken from first generation when 4 threads are applied (p=1, N=100000, g = 30, r = 0.0001, m= 0.00025)

Rys. 4. Czas wykonywania programu symulacyjnego dla różnego rozmiaru próbki pobieranej z pierwszej generacji, gdy wykorzystywane są 4 wątki (p=1, N=100000, g = 30, r = 0.0001, m= 0.00025)

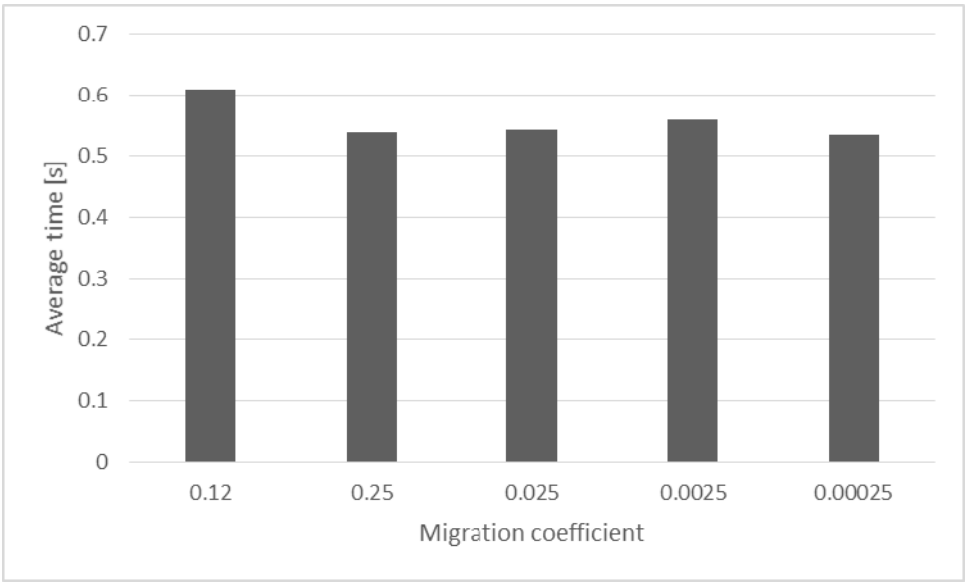

Fig. 5. Time of simulation application execution for various values of migration parameter when 4 threads are applied (p=2, n=6000, N=100000, g = 30, r = 0.0001)

Rys. 5. Czas wykonywania programu symulacyjnego dla różnych wartości współczynnika migracji, gdy wykorzystywane są 4 wątki (p=2, n=6000, N=100000, g = 30, r = 0.0001)

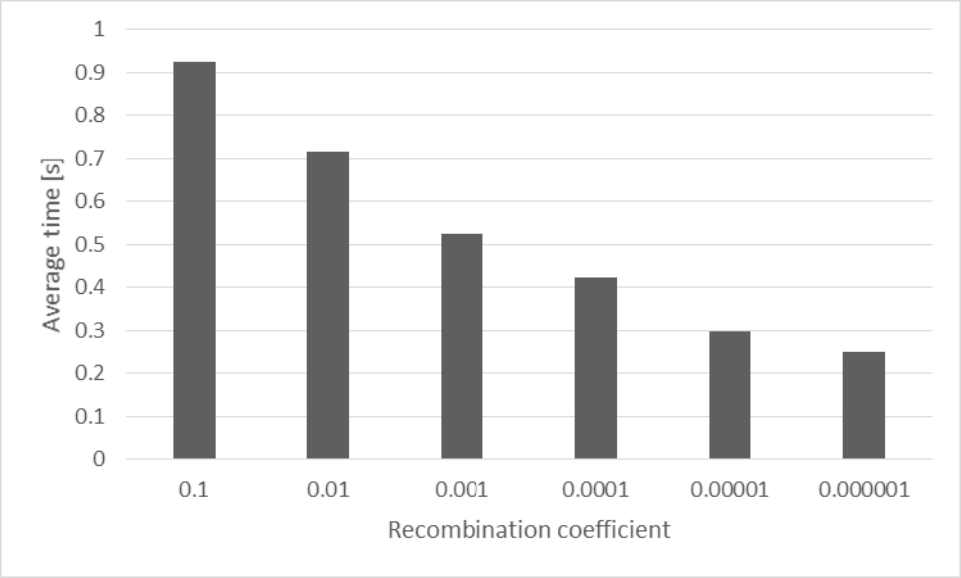

Fig. 6. Time of simulation application execution for various values of recombination parameter when 4 threads are applied (p=1, n=6000, N=100000, g = 30, m= 0.00025)

Rys. 6. Czas wykonywania programu symulacyjnego dla różnych wartości współczynnika rekombinacji, gdy wykorzystywane są 4 wątki (p=1, n=6000, N=100000, g = 30, m= 0.00025)

4. Discussion

Utilization of multithread approach with shared memory platform allowed for achievement of significant performance gain (fig. 2). As usually in case of parallel application increase of the number of computation cores does not end up in linear reduction of result await time, what is more when the data are analysed concurrently, in too many computational units, there is a possibility of degradation of processing speed.

The best number of involved computational cores depends on the number of effective population size and the number of sampled individuals, with the increase of the number of effective population size, increase of the number of computational cores is suggested.

5. Summary

Ongoing researches in the area of biology requires creation of efficient simulation software which is able to generate huge amount of data in short time. Proposed improvements, in the area parallel processing allowed for significant reduction of result await time and better

utilization of computational power of recent computers. Importantly performance gain was stable when various simulation coefficients were modified

Computers equipped in multicore processors are widely available, therefore application of parallel approach is important for proper utilization of resources. Such platforms have many advantages such as shared memory which allows for achievement of easy communication between threads, on the other hand shared memory might be limiting factor, therefore if further gain in computational speed is needed application of high performance computing based on processes communication might be considered.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the PEOPLE Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme FP7/2007-2013/ under REA grant agreement no. 298995. Krzysztof Cyran has been supported by the above mentioned grant, Dariusz Myszor has been supported by BK-215/Rau2/2013.

BIBLIOGRAPHY

1. Ren B., Robert F, Wyrick J. J., Aparicio O., Jennings E. G., Simon I., Zeitlinger J., Schreiber J., Hannett N., Kanin E., Volkert VOL. L., Wilson C. J., Bell S. P., Young R. A.: Genome-Wide Location and Function of DNA Binding Proteins, Science, 2000, vol. 290, no. 5500, p. 2306÷2309.
2. Mukherjee S.: Rapid analysis of the DNA-binding specificities of transcription factors with DNA microarrays. Nature Genet., 2004, vol. 36, p. 1331÷1339.
3. Smith R., Mathis A. D., Ventura D., Prince J. T.: Proteomics, lipidomics, metabolomics: a mass spectrometry tutorial from a computer scientist's point of view. BMC Bioinformatics, 2014, no. 15 (7), p. 9.
4. Palsson B.: The challenges of in silico biology. Nature Biotechnology, 2000, vol. 18, p. 1147÷1150.
5. Butcher E. C, Berg E. L., Kunkel E. J.: Systems biology in drug discovery. Nature Biotechnology, 2004, vol. 22, p. 1253÷1259.
6. Lua X., Wanga H., Wanga J., Xub J., Lia D.: Internet-based Virtual Computing Environment: Beyond the data center as a computer. Future Generation Computer Systems, 2013, vol. 29, no. 1, p. 309÷322.

7. Abouelhoda M., Issa S., Ghanem M.: Towards Scalable and Cost-aware Bioinformatics Workflow Execution in the Cloud - Recent Advances to the Tavaxy Workflow System, Journal Fundamenta Informaticae, 2013, vol. 128, no. 3, p. 255÷280.
8. Chang V., Walters R. J., Wills G.: Cloud Computing and Services Science. Communications in Computer and Information Science, 2013, vol. 367, p. 245÷264.
9. Ewens, W. J.: Mathematical Population Genetics, Springer, Berlin 1979.
10. Agrafioti I., Stumpf M. P. H.: SNPSTR: a database of compound microsatellite-SNP markers. Nucleic Acids Res., 2007, vol. 35, supplement 1, p. D71÷D75.
11. Cyran K. A., Myszor D.: Coalescent vs. time-forward simulations in the problem of the detection of past population expansion, International Journal of Applied Mathematics and Informatics, vol. 2, no. 1, 2008, p. 10÷17.
12. Kingman, J. F. C.: Origins of the coalescent: 1974–1982. Genetics, 2000, vol. 156, p. 1461÷1463.
13. Hein J., Schierup M., Wiuf C.: Gene Genealogies, Variation and Evolution: A Primer in Coalescent Theory. Oxford University Press, 2004.
14. Donnelly P., Tavare S.: Coalescents and genealogical structure under neutrality. Annual Reviev Genetics, 1995, vol. 29, p. 401÷421.
15. Hudson R. R.: Gene genealogies and the coalescent process. Oxford survey evolutionary biology, 1990, vol. 7, p. 1÷44.
16. Kingman, J. F. C.: The coalescent. Stochastic Process. Appl., 1982, vol. 13, p. 235÷248.
17. Liang L., Zollner S., Abecasis G. R.: GENOME: a rapid coalescent-based whole genome simulator. 2007, Bioinformatics, vol. 23, no. 12, p. 1565÷1567.
18. Matsumoto M., Nishimura T.: Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Transactions on Modeling and Computer Simulation, 1998, vol. 8, p. 3÷30.

Omówienie

Artykuł prezentuje modyfikacje wprowadzone do GENOME: A rapid coalescent-based whole genome simulator, których zadaniem jest przyśpieszenie procesu uzyskiwania wyników. Modyfikowana aplikacja bazuje na jednym procesie wykonującym pojedynczy wątek, dlatego też postanowiono wprowadzić wielowątkowość, która powinna pozwolić na lepsze wykorzystanie współcześnie dostępnych zasobów obliczeniowych oraz uzyskanie skrócenia czasu oczekiwania na wyniki. W artykule przedstawiono możliwe modyfikacje oraz opisano zaimplementowane rozwiązanie. Opisano także problem inicjalizacji generatora liczb pseu-

dolosowych w oprogramowaniu wielowątkowym oraz przedstawiono zaimplementowane rozwiązanie bariery synchronizacyjnej.

Wyniki przeprowadzonych badań wykazały, iż w wyniku wprowadzenia wielowątkowości można uzyskać znaczne skrócenie czasu wykonywania aplikacji (nawet o 40%).

Addresses

Krzysztof CYRAN: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16
44-100 Gliwice, Polska, krzysztof.cyran@polsl.pl.
Dariusz MYSZOR: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16
44-100 Gliwice, Polska, dariusz.myszor@polsl.pl.