УДК 681.513.8

PARALLEL RELAXATIONAL ITERATIVE ALGORITHM OF GMDH

Andrey Pavlov

International Research and Training Center of Information Technologies and Systems of National Academy of Sciences of Ukraine, 03680, Kiev, av. Glushkova, 40, Ukraine

andriypavlove@gmail.com

У роботі запропонована розпаралеленаверсія узагальненого релаксаційного ітераційного алгоритму з рекурентними обчисленнями. Розпаралелення за допомогою бібліотеки TreadingBuildingBlocks дозволило прискорити алгоритм в 2.28х на 2 ядрах і в 4х на 4 ядрах процесора Intel, що використовує технологію Hyper-Treading.

Ключові слова: TBB, OpenMP, багатопотокове розпаралелювання, узагальнений релаксаційний ітераційний алгоритм, метод групового урахування аргументів.

The paper suggests a parallel version of the generalized relaxational iterative algorithm with recurrent computations. Parallelization using TBB library allow getting 2.28x speedup using 2 cores and 4x speedup using 4 cores of Intel processor with hyper-threading technology.

Key words: TBB, OpenMP, multithreading parallelization, generalized relaxational iterative algorithm, group method of data handling.

В работе предложена распараллеленная версия релаксационного итерационного алгоритма с рекуррентными вычислениями. Распараллеливание с помощью библиотеки TreadingBuildingBlocks позволило получить ускорение в 2.28х на2 ядрах и в4х на 4 ядрах процессора Intel, использующего технологиюHyper-Treading.

*Ключевые слова:*TBB, OpenMP, многопотоковое распараллеливание, обобщённый релаксационный итерационный алгоритм, метод группового учёта аргументов.

Introduction

Multicore processors aren't new today.Probably every modern computer is equipped with such a central processing unit today. This technical progress make engineers, scientists and programmersreorganize their way of thinking anddeveloping algorithms and methods according to principles of parallel programming.The algorithms of the group method of data handling (GMDH) are highlytime-consuming but can be easily parallelized.

An analysis of approaches to a parallelization of known iterative GMDH algorithms was carried out in [1]. The [1] suggests new parallelization principles for one of the fastest iterativeGMDH algorithm – generalized relaxational iterative algorithm (GRIA) [2] and locates the most resource-intensive algorithm sections that are required to be parallelized.

This paper describes the problemsthat authorencountered when implementinga parallel version of the algorithm and shows the results of the GRIA scalability.

The algorithm is written in C++, therefore, let us first, present a survey of today's C++ libraries for multithread parallelization.

1. Overview of libraries for parallelization

The most well-known libraries are: Message Passing Interface (MPI) [3], Open Multi-Processing(OpenMP) [4] andThreading Building Blocks (TBB) [5].

MPI is an API developed for message transferring thatallow exchange messages between processes which perform one task. Primarily the interface designed for parallelizing of tasks between CPUs but not threads, therefore creating, deleting, synchronizing and message transfer between threads shoulder a programmer. That is the reason the interface is a quite low-level and complex, and isn't easy to master in a short term.

OpenMPis an open standard for program parallelization describing a set of compiler directives, library procedures and environment variables which purposed for creating multithread applications based on shared memory multiprocessor systems [6]. The standard is implemented in the majority of well-known compilers (GCC, Microsoft Visual Studio, Intel, IBM, Oracle) and allows writing parallel applications easily just by parallelizing local code segments adding minimum changes to the original code.

The library automatically manages threads and distribute tasks over the treads. A programmer should only specify how exactly the tasks must be split between threads. The standard has several parameterized ways for loop parallelization which can be successfully used in a certain situation. The easiness of application of this library, clarity of the final parallel code and simplicity of making changes arethe significant advantages over the MPI.

TBBis a C++ template library for parallelism undertaking thread management that allow (as OpenMP) directly specify the section of code which should be parallelized. A programmer should think in terms of tasks, not threads when applying the library.

Unlike OpenMP, TBB automatically form packages of tasks which will beprocessedin corresponding threads. TBBbalances a workload of all available processor cores as evenly as it is possible. This feature is a great advantage of the library over other libraries. The mechanism consists in splitting overall set of the tasks in subsets or packages (and assigningthem to the threads) until the workload of the cores will be even. If a thread finished processing its tasks and there is a thread that still have unprocessed tasks, TBB balances total workload using its internal task stealing mechanism. As OpenMP, the library has all necessary facilities for loop parallelization. The advantages of TBB over OpenMP are:

- any data type support;
- ready to useclass templates, allowing memory access from several threads simultaneously;
- automatic detection and creation the optimal number of working threads in runtime.

The library is ultimately easy to use and can be modified easily to any specific tasks due to open source. Let us compare the performance of OpenMP and TBB in taskof parallel computation of normal matrix – one of the most time-taking stage in the GRIA.

2. Performance comparison of OpenMPand TBB

Implementation of the normal matrix calculation in [1] consists in the execution of nested loops where the limits of the inner loop depend on the limits of the outer. This dependence prevents from parallelizing the body of the inner loop. Both libraries require independence of iterations in a loop. Therefore the nested loop was replaced by an equivalent single loop that allow to speed up the procedure in 3.5x, even in the serial version: the normal matrix$\mathbf{X}^T\mathbf{X}$ (dim $\mathbf{X}$ is 5000 observations and 1000 variables) was calculated using the nested loop in 1 min 51 Sec and using the single loop in 31 Sec.

Performance comparison was carried out when calculating a normal matrix$\mathbf{W} = \mathbf{X}^T\mathbf{X}$, dim $\mathbf{X} = 5000 \times 1000$. The Intel Core i3 M 350 2.27GHz processor having 2 physicaland 2 logical (due to hyper-threading technology)cores was used. The calculation was repeated 20 times. Figure 1 shows the average time of the matrix calculation.

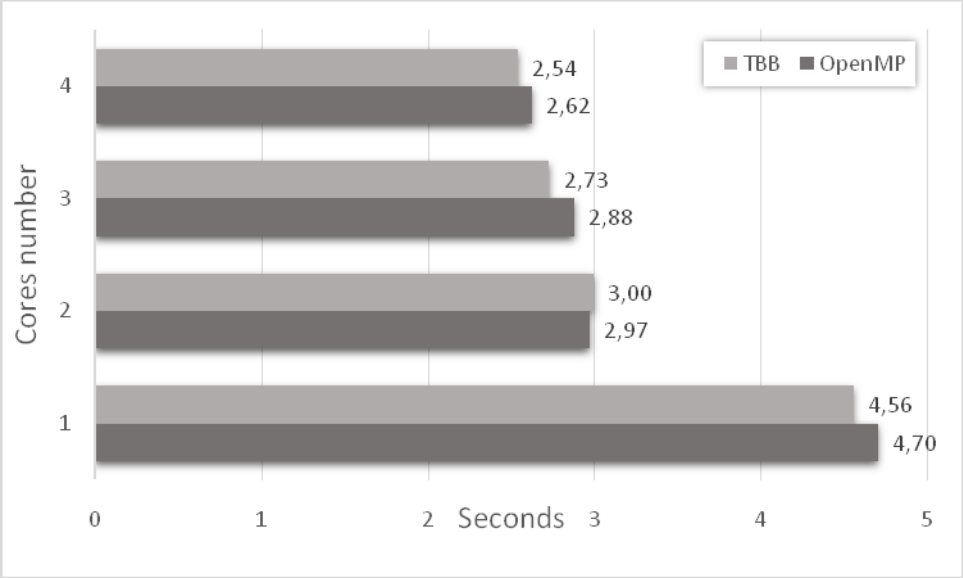

Figure 1. Normal matrix calculation time using paralallization

As one can see, the calculation time when using the TBB is slightly less than in case of using OpenMP.Should be stressed that the usage of two additional logical cores hardly speed up the procedure, indicating that parallelization such a low-level operations as multiplying and summing use the resources of the two physical coresalmost completely.

The decision about using the TBB library for parallelization the GRIA was taken, rooting in the results of the experiment and taking into account the advantages of the TBB mentioned above.

3. Parallel relaxational iterative GMDH algorithm

The generalized relaxational iterative algorithmhastwo model structure generators: exhaustive search generator and directed search generator. Each of the generator

defines a separate relaxational algorithm. This work as the [1] focuses on the parallelization the algorithm based on the generator of exhaustive search.

According to [1], there can be allocated two the most time-consuming stages in the algorithm: normal matrix calculation stage (is a common stage for every algorithm in GRIA) and stage of iterations where models are built.

After the models have been built using the GRIA, they must be evaluated: models' response anddifferent kinds of errors should be calculated on training, validation and holdout sets. This process, called *Errors calculation*, turned to be quite time-taking if the length of the sets is long and the *F*choice freedom parameter is great. For example, if *F* is 400, the iterations number is 50 and the data length is 5000 observations, then the calculation oferror measures for 20,000 models takes near 15 sec. And the longer is the data length, the longer is the time to perform this stage.

3.1 Parallelization of the normal matrix calculation stage

A scheme for generation and distribution the tasks over available computing units (threads) on the normal matrix calculation stage was suggested in [1]. The scheme was developed to balance the workload of the threads in case of using the nested loop. The previous section of this paper says about the requirements in parallelizing loops via TBB, especially of the independence of loop iterations. To satisfy this condition, a new single loop was suggested that is equivalent to the nested one.This single loop is the subject of parallelization here.

Due to automatic detection of optimal volume of tasks packages (number of loop iterations) which will be processed in parallel and distribution/balancing of the work-load between threads, a programmer shouldn't carry about that.

There were parallelized the loops for calculation of the $X_A^T X_A$, $X_B^T X_B$ matrices and $X_A^T y_A$, $X_B^T y_B$ vectors in this stage.Scalars $y_A^T y_A$, $y_B^T y_B$ are left calculated serially.

3.2 Parallelization of the stage of iterations

The loop of building the models in serial version of the algorithm looks like:

```
while ( Model *model = generator->next() )
{
estimator->estimate( model );
criterion->calculate( model );
if ( !selector->select( model ) )
 {
 delete model;
 }
}
```

TBB has tools for parallelization of `while` loop, but the bottleneck is that only one thread at a time can access the method of the model generation `generator->next ()`. This fact prevents gaining a good scalability of the stage. Therefor we decide to reimplement the loop in a way that the bottom and upper bounds of the loop are known.

The idea is to create a separate model structure generator for every package of tasks and make it generatea particular set of model structure such that the sets of any two generators will be nonoverlapping and the union of all the sets give the original total set. To avoid simultaneous access to the methods `estimate` and `calculate` of the objects `estimator` (estimation of model parameters) and `criterion` (calculation of the criterion),an independent copy of these objects should be created for every package of tasks.

The second task of parallelization of the loop is the method of model selection `select`. The object `selector` contains a map of models ranked by minimizationof criterion value. Once a model haslessercriterion value than the last in the map has been added, the last model is removed from the map. This situation causesaccess violation errors if one thread has already deleted the last model from the map and another one requests information about this model because the size of the map is not updated yet up to this moment from the first thread.

There are two ways to resolve this problem: 1) synchronization of the threads when accessing the `select` method of the `selector` object; 2) creation of independent copies of the `selector` object for every tasks package. The extraordinarily small time of model building causes a high access frequency to the `select` method, which will be a bottleneck as in case with parallelizing the `while` instruction.Thus, let us concentrate on the second way.

Despite that the second way allows getting completely independent loop iterations, it has a disadvantage however. If the copies of the objects `generator`, `estimator` and `criterion` require a little volume of RAM, the `selector` stores F selected models. Each copy of the `selector` object must store map of F models to guarantee that the result of the serial version of the algorithm will not be lost. This demand significantly increases consumption of RAM if the number of the packages is large. Actually, expenditure of RAM increase as many times as number of packages were generated. It is suggested to directly set the (minimal) number of packages equal to the number of working threads, to minimize RAM consumption. This assignment (restriction) prevent sometimes to get even work-load of the cores and is a drawback of the suggestion. Achievement the balance of cores work-load by increasing the number of the packages up to the optimal level is one of the basic tasks of TBB.

3.3 Parallelization of the errors calculation stage

The procedure of model errors calculation consists of a loop with a known range. This loop uses the `errorsCalculator` object that calculates errors on training, validation and holdout sets. The parallelization is realized similarly to the normal matrix calculation stage, by creating copies of `errorsCalculator` object for every package of tasks.

4 Scalability test of the parallel version

Let us compare time to run serial version and parallel under different number of available cores. A test was conducted using two processors:

- Intel Core i3 M 350 2.27GHz processor with 4cores (2physicaland2logical);
- Intel Core i7 4700 HQ 2.4GHz processor with 8cores (4 physicaland 4 logical).

The input matrix **X** contained 4000 observations, 10 true variables and 990 false variables. It was generated using pseudo random generator Mersennetwister [7]. Values were generated in the [0; 1] interval by uniform distribution law. The set of observations was divided into holdout set of 500 observations, training set of 3000 observations and validation set of 500 instances. Parameters of the algorithm such as choice freedom F and the number of iterations R were set as 400 and 50 correspondently. The algorithm should findthe model

$$y = 6.29447 + 9.37736x_1 + 8.26752x_2 - 8.04919x_3 + 8.11584x_4 - 7.46026x_5 -$$

$$-7.29046x_6 + 6.70017x_7 - 5.57932x_8 - 3.83666x_9 + 2.64719x_{10}.$$

The algorithm has found the model

$$\hat{y} = 6.29447 + 9.37736x_1 + 8.26752x_2 - 8.04919x_3 + 8.11584x_4 - 7.46026x_5 -$$

$$-7.29046x_6 + 6.70017x_7 - 5.57932x_8 - 3.83666x_9 + 2.64719x_{10} + 3.5 \cdot 10^{-8} x_{145} +$$

$$+1.4 \cdot 10^{-9} x_{258}.$$

Figures 2 and 3 show the results of algorithm scalability bythe three algorithm stages mentioned above.

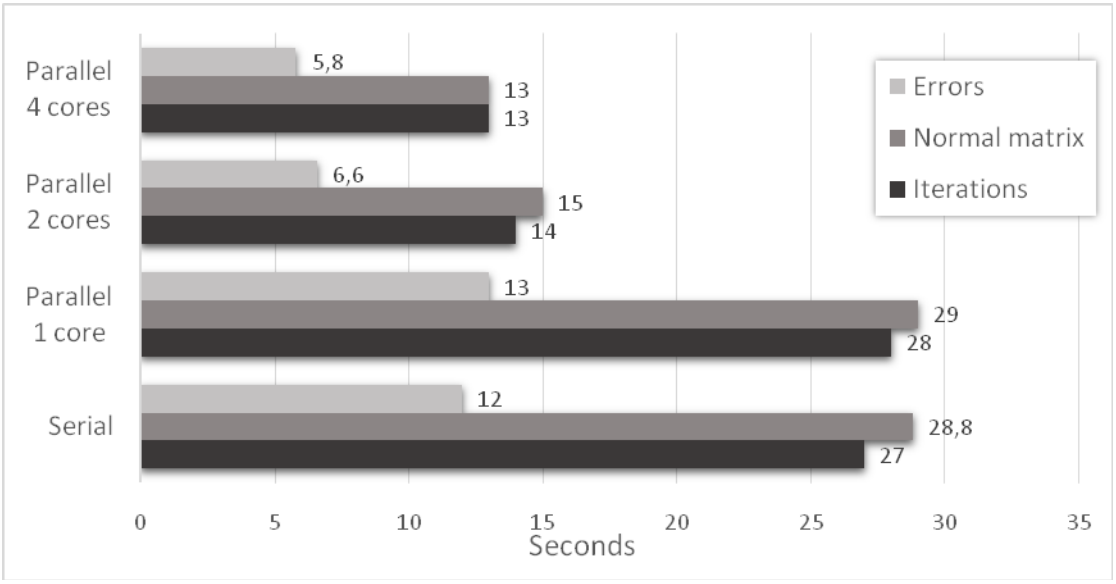

Figure 2. Scalability of the algorithm stages using Intel Core i3

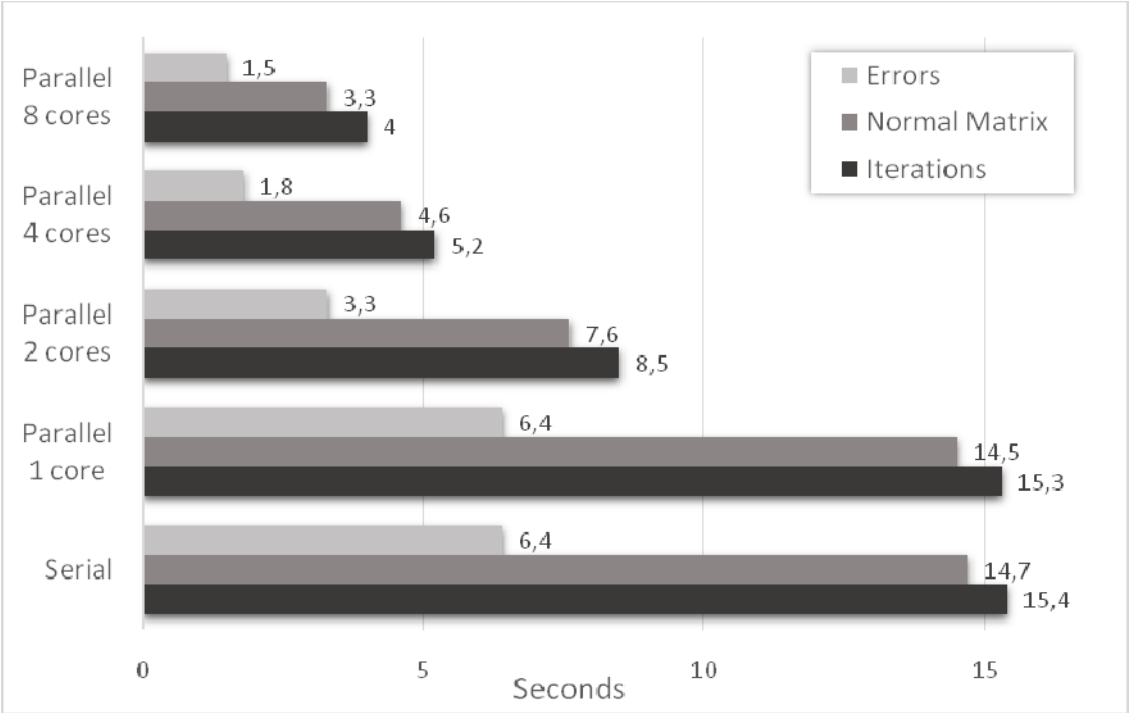

Figure 3. Scalability of the algorithm stages using Intel Core i7

As you can see from the figures, TBB allows obtaining almost equal time of the serial and the parallel version using one core.Note that, usage of physical cores allows speeding up every stage in 2x using 2 cores of Core i3 and in 1.86x using 2 cores of Core i7. Scalability of the stages differs when using 4 cores of Core i7: the iterations stage has been speeded up in 2.9x, the normal matrix calculation stage – in 3.15x and the errors calculation stage – in 3.55x.

The scalability is week when adding logical cores what isin concordance with the results of libraries performance tests: usage of 4 cores of Core i3 speedups the algorithm only in 2.3x, and usage of 8 cores of Core i7 – 3.8x for the iterations stage, 4.39x for the normal matrix calculation stage and 4.26x for the model errors calculation stage.

Let’s look at the scalability of the whole algorithm (fig. 4).

As you see, TBB speedups the algorithm in:

- 2xusing 2 cores and 2.28x using 4 cores of Core i3;
- 1.86xusing 2 cores, 3xusing 4 coresand 4xusing 8 cores of Core i7.

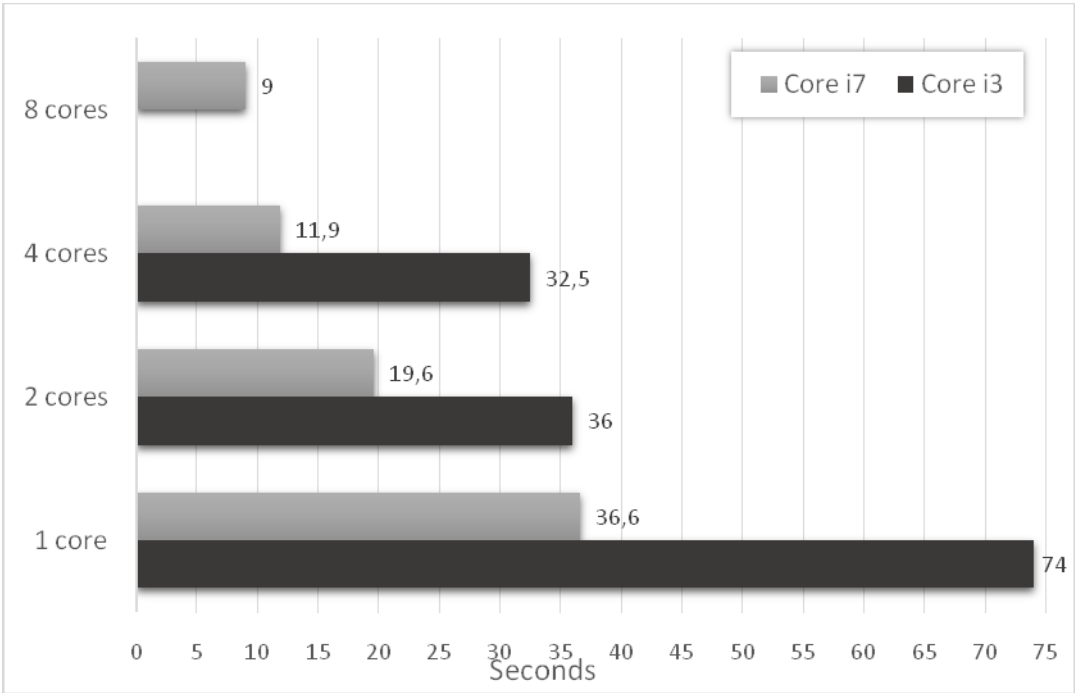

Figure 4. Algorithm scalability

5 Conclusion

Speeding up the algorithm in 2.28x and 4x is even more than ideal result, taking into account that processors Core i3 and Core i7 have only two and fourphysical corescorrespondently.

References

1. Pavlov A.V. Principles of parallel computations of relaxational iterative GMDH algorithm /«Inductive modeling of complex systems», collection of sc.works, № 5 – K.: IRTCITS, 2013. – P.220-225. (in Russian)

2. PavlovA.V. GeneralizedrelaxationaliterativeGMDHalgorithm // Inductive-modelingofcomplexsystems. Col. sc. works, iss. 2. – K.: IRTCITSNASU, 2011. – P. 95-108. (in Russian)

3. Internetresource http://en.wikipedia.org/wiki/Message_Passing_Interface

4. Internetresource http://openmp.org/wp/

5. Internetresource https://www.threadingbuildingblocks.org/

6. Internetresource http://en.wikipedia.org/wiki/Symmetric_multiprocessing

7. Internetresource http://en.wikipedia.org/wiki/Mersenne_twister