UDC 004.438:519.685.8
DOI: 10.20535/SRIT.2308-8893.2022.2.03

COMPARATIVE ANALYSIS OF THE EFFECTIVENESS OF USING FINE-GRAINED AND NESTED PARALLELISM TO INCREASE THE SPEEDUP OF PARALLEL COMPUTING IN MULTICORE COMPUTER SYSTEMS

V. MARTELL, A. KOROCHKIN, O. RUSANOVA

Abstract. The article presents a comparative analysis of the effectiveness of using parallelism of varying granularity degrees in modern multicore computer systems using the most popular programming languages and libraries (such as C#, Java, C++, and OpenMP). Based on the performed comparison, the possibilities of increasing the efficiency of computations in multicore computer systems by using combinations of medium- and fine-grained parallelism were also investigated. The results demonstrate the high potential efficiency of fine-grained parallelism when organizing intensive parallel computations. Based on these results, it can be argued that, in comparison with more traditional parallelization methods that use medium-grain parallelism, the use of separately fine-grained parallelism can reduce the computation time of a large mathematical problem by an average of 4%. The use of combined parallelism can reduce the computation time of such a problem to 5,5%. This reduction in execution time can be significant when performing very large computations.

Keywords: multicore computer system, core, thread, tasks, parallelism, granularity, fork-join, speedup coefficient, fine-grained parallelism, nested parallelism, combined parallelism.

INTRODUCTION

Classical von Neumann architecture is not designed for parallel computing; all commands execute in one sequence strictly one after another. Each such individual sequence that operates on the machine at the current time is a process. ISO 9000:2000 [1] defines a process as a set of interconnected and interacting actions that convert input data into output. A computer program in itself is only a passive sequence of instructions, while a process is the direct execution of those instructions.

The concept of process is inextricably linked with the concept of thread. The execution thread is the smallest unit of processing, the execution of which can be assigned by the operating system kernel [2]. Implementation of execution threads and processes in different operating systems differs from each other, but in most cases, the execution thread is within the process. Multiple threads can exist within the same process and share resources such as memory, while processes do not share these resources. In particular, the execution thread hare process instructions (its code) and its context (the values of the variables they have at any given time).

Thus, in the framework of von Neumann or single-core architecture, parallelism is usually realized by time multiplexing: the processor switches between different threads. This context switching is called pseudo-parallelism and usually occurs often for the user to perceive the execution of threads or tasks as simultaneous [3]. In such systems, there are effective time schedulers in standby mode (blocked) [4]. Scheduling is based on the principle of priorities, for example, in most cases, some user input has a higher priority than some computations, so if the central processor unit (CPU) receives an input signal, a priority interrupt occurs and the CPU processes its, which allows the user to control most of process, such as urgently terminate some of them.

However, in the present stage, this approach no longer ensures compliance with the requirements for computer systems. And with the emergence of multi-core and multiprocessor architecture, the question arose of the organization of real parallelism, when at the point of time each individual core or processor performs its own thread. In multiprocessor and multi-core systems, threads or tasks can run simultaneously, with each processor or core processing a separate thread at the same time. The operating systems of such computers are much more complex and voluminous because, for such architectures to function effectively, their components must also exchange data (communicate and synchronize), and do so in a timely, fast, and with minimal computational downtime. Planners of such operating systems plan the distribution of captures by the thread of cores both in time and space [4]. So, even parallel threads are not all the same. In addition to the level of their priority, you can enter another measure for them – quantitative, one that is based on the volume of the basic unit in the program.

PROBLEM ANALYSIS AND TASK STATEMENT

Each individual elementary parallel computation in modern multi-core computer systems (MCS) can be represented as a granule. Depending on how many such elementary calculations (parallel within the system, but consecutive within one thread) each thread contains, and how many communications (i.e. interruptions of parallelism) were conducted between such threads, we can introduce the concept of "granularity".

Granularity is a measure of the ratio of the number of calculations performed in a parallel problem to the number of communications [5]. The degree of granularity varies from fine-grained to coarse-grained.

It should be noted that the granularity classifications available in different sources differ slightly, especially for coarse- and medium-grained parallelism, so the following classification will be taken as a basis in this article:

1. Coarse-grained parallelism: each parallel calculation is quite independent of the others, and requires a relatively rare exchange of information with other calculations. The units of parallelism are large and independent programs that include thousands of commands [6].

2. Medium-grained parallelism: units of parallelization are individual procedures that are called in separate threads and include hundreds of commands. It is usually organized by both the programmer and the optimizing compiler. Most general-purpose parallel computers are primarily focused on this category of parallelism [7].

3. Fine-grained parallelism: each parallel calculation is quite small and elementary, consists of dozens of commands. Usually, the units that are parallelized are elements of expressions or individual iterations of a loop. They usually have

little or no relationship between the data. The amount of work associated with a parallel task is low and the work is evenly distributed among the processors. Hence, fine-grained parallelism facilitates load balancing [8].

The very term “fine-grained parallelism” refers to the simplicity and speed of any computational action. A characteristic feature of fine-grained parallelism is the approximate equality of computational intensity and data exchange. This level of parallelism is often used by the parallelizing (vectorizing) compiler [9], as well as recently in asynchronous programming. This work focuses on medium- and fine-grained parallelism.

Fine-grained parallelism has a long history: it is the most “ancient” kind of parallelism. The development of his theory took place simultaneously with the development of the theory of successive calculations and is associated with the name of the already mentioned John von Neumann. His theoretical model of a calculator with fine-grained parallelism is widely known – “cellular automaton” [10]. But, when in the process of development of computer technology there were opportunities and capacities for the organization of full-fledged threads of medium-grained and coarse-grained parallelism, there was some decline in interest in fine-grained parallelism. However, the decline in interest changed when technological advances, on the one hand, led to the fact that medium- and coarse-grained parallels somewhat exhausted their significant development, and on the other hand allowed to create a unified architecture within which at different stages of program implementation could move on parallelism of various degree of granularity, i.e. on some parts of programs to use the classical mechanism of threads, and on others fine-grained parallelisms, like tasks or parallel loops.

Against the background of renewed interest in fine-grained parallelism in modern programming languages and libraries, along with the tools for organizing parallel computations by creating a set of classic full-fledged medium-grain threads, there are tools for computing within fine-grained parallelism. Most often, these are tools for parallelizing loops or small sets of similar commands.

However, it also led to conflicts in the design stages of modern parallel software. Currently, there are two opposing approaches to creating the architecture of such software: on the one hand, you can represent the parallel part of the program in the format of medium-grained structures (classical threads), on the other hand, all calculations can be represented in the form of small tasks. At the same time, if we can say that the classical concept of threads is not very suitable for modern back-end problems, in contrast to the concept of tasks, then in the field of high-load and intensive scientific calculations, solving classical problems can be presented using both threads and tasks.

At the same time, given the above-described nature of parallelism in modern MCS, as well as modern flexible tools for organizing different levels of parallelism in languages and libraries of parallel programming, we can assume that there is space and opportunities for effective solving of computational problems with simultaneous use of both mechanisms. This combination is called “Nested parallelism”.

Therefore, the purpose of this work is to research the possibilities of improving the execution time of parallel programs in MCS through the use of fine-grained parallelism, the optimal combination of medium and fine-grained parallelism (nested parallelism), as well as the use of modern software instruments of their implementation.

FINE-GRAINED PARALLELISM IMPLEMENTATIONS OVERVIEW

Among the most popular languages and libraries currently used in high-load computing, fine-grained parallelism tools are implemented in the OpenMP library (which is usually used in combination with C++), Java and C# languages [11–14].

OpenMP

Fine-grained parallelism is represented by a special preprocessor directive **#pragma omp parallel for**, which refers to the work-sharing directives. Such directives are not used only for parallel code execution; they are used for the logical distribution of a group of threads to implement these control logic constructs. The #pragma omp for directive informs that when running a loop in parallel mode, loop iterations must be distributed between a group of threads. Execution of the following program code:

1. int size=100; //the number of calculation iterations
2. #pragma omp parallel for
3. for(int i = 0; i < size; i++)
4. Calculations();
5. ShowResults();

in the four-processor system would happen as shown in Fig. 1. This distribution is used by default and is called static scheduling.

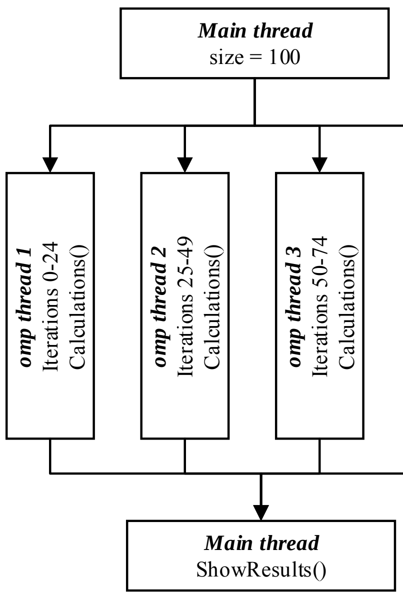

Fig. 1. The scheme of distribution loop iterations in parallel threads using OpenMP

OpenMP also provides another, more flexible types of scheduling, such as dynamic scheduling, runtime scheduling, and guided scheduling. The special section is used to set up one of these modes. The following code scheme shows the format of this section: **schedule *(algorithm name)* num_threads *(number of iterations)***. This mechanism is used when different iterations perform different amounts of work and determine whether a thread that has already completed its iteration will be able to take over part of the work of another thread.

When writing more complex parallel programs, one of the key tasks to solve is the problem of thread synchronization. In OpenMP, implicit barrier synchronization is at the end of each #pragma omp parallel and parallel for block. There are also manual synchronization tools, such as barriers that can be created using the #pragma omp barrier directive.

With writing the parallel program, the second main task that needs to be figured out is the mutual exclusion problem, which avoids the situation so-called "race condition", when a large number of threads will behave unpredictable or be blocked due to access to the same area of memory, which, for example, contains some variable (shared resource), which used in all threads. With the large number of threads created with the application of fine-grained parallelism, this problem is particularly acute. OpenMP provides the ability to use special nonblocking tools to solve the problem of mutual exclusion, such as the directive #pragma omp atomic, as well as modifiers of the directive #pragma omp parallel for: shared, private, firstprivate, lastprivate or reduction. Manual means of solving the problem of mutual exclusion are represented by such constructions as locks and critical sections.

Java

Developers are offered extremely flexible and powerful tools for implementing fine-grained parallelism based on the Fork-Join model, realized in built-in package **java.util.concurent**.

A recursive algorithm is used to implement this model in Java, which described in the following paragraph:

1. The check on the possibility of dividing the actions of this thread into two smaller tasks.
2. If the check is successful, the distribution is performed (Fork), by creating new threads for each new task. In each new thread, the algorithm begins anew. The thread that performed the distribution is blocked until both created threads have finished their work, and then it performs the final collection of the result.
3. If the check is not successful (the limit of the so-called "grain of parallelism" is reached), then the calculations are performed in this thread, after which the connection with the generated thread occurs (Join).

This implementation contains two classes: **RecursiveAction** and **RecursiveTask**. When it is necessary to calculate a specific numeric value of a large function (such as the sum of vector elements), it is better to use Recursive Task. In the case of general operations, the results of which are not a specific number, RecursiveAction works better. By inheriting these classes and describing the computer's own overload, the developer can customize the work to solve a specific problem, which is extremely effective.

An example of using the RecursiveAction class to implement fine-grained parallelism in Java organizing a parallel loop as an example is shown in the listing below:

```
class ParallelFor extends RecursiveAction {
    private int from, to;
    volatile final int GRAIN = 25;
    public ParallelFor(int from, int to) {
        this.from = from;
        this.to = to;
    }
    protected void compute() {
        int len = to - from;
        // Stop condition of main recursion
        if (len < GRAIN)
            work(from, to);
        else {
            int mid = (from + to) >>> 1;
            ForkJoinTask<Void> parallelFor1 =
                = new ParallelFor(from, mid).fork();
            ForkJoinTask<Void> parallelFor2 =
                = new ParallelFor(mid, to).fork();
            parallelFor1.join();
            parallelFor2.join();
        }
    }
}
// Parallel loop startup is performed using the invoke()
// method called on the instance of the ForkJoinPool class
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new ParallelFor(from, to));
```

The variable *GRAIN* determines the depth of the partition, in other words, the amount of work, after achieving which, the thread will start to perform it, rather than making further parallelization. The *work()* method may contain certain basic parameterized calculations, which performance is expected from each Fork-Join task after reaching the maximum depth of parallelization. For example, after setting the initial values of the variables *GRAIN* = 25, *from* = 0, *to* = 100, we obtain a parallel execution of the loop of 100 iterations discussed in the previous subsection, in which each of the Fork-Join tasks will receive 25 iterations for processing. However, unlike OpenMP, the structure of the generated threads and their control hierarchy will be treelike, as shown in Fig. 2.

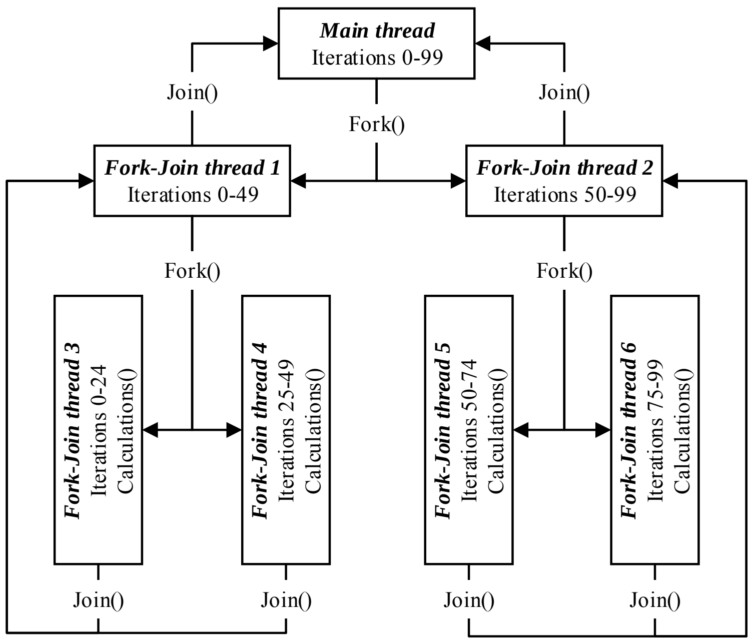

Fig. 2. The scheme of distribution loop iterations in Fork-Join parallelization using Java

The resulting structure is more complex than the one in OpenMP. Nevertheless, according to the results provided in the next chapter, such an approach to fine-grained parallelism proved to be no less effective.

C#

C# also provides the ability to implement fine-grained parallelism. All the necessary functionality for this is contained by a static class **System.Threading.Tasks.Parallel**, namely by its three main methods **Parallel.For()**, **Parallel.ForEach()**, **Parallel.Invoke()** and their various overloads. Parallel.For and Parallel.ForEach provide parallel execution of for and foreach loops, respectively. The override methods presented in this class are aimed to maximize the parameterization of parallelism, depending on the specific task being implemented.

Each method described above is based on a mechanism similar to the one used in Java. Nonetheless, it has also been simplified with a mechanism for delegating and anonymous methods. Developers are not required to manually write the entire recursive part and the regulation of grain parallelism, this is the responsibility of the execution environment. Therefore, in practice, the implementation of fine-grained parallelism becomes extremely simple and has almost no different from the classical single-threaded approach.

The following piece of code provides a similar parallelization to the previous loop of a hundred iterations:

```
Parallel.For(0, 100, i => {
    Calculation();
});
```

Although parallelization in C# occurs by the very same mechanism as in OpenMP, through the simplifying, that assures the virtual environment of the CLR execution, the resulting threads' structure and their hierarchy of management will be similar to that in the OpenMP library.

Also in C# are realized the tools for the organization of fine-grained parallelism manually, similar to corresponding tools in Java. The *Task* class (**System.Threading.Tasks**) is responsible for this. The formed tasks can be performed in one or more threads. The official documentation of this programming language recommends using them primarily for asynchronous programming. In fact, the tools for organization parallel loops in C# discussed above are high-level abstractions constructed using Tasks.

NESTED (COMBINED) PARALLELISM

This approach is based on the use of two types of parallelism in the parallel program: medium-grained and fine-grained. The program includes a set of traditional threads for the number of MCS cores. Each of these threads additionally implements internal (fine-grained) parallelism by creating subthreads using appropriate Fork-Join tools. The initial number of traditional threads can be reduced in order to provide fine-grained parallelism with free processor resources. A parallel thread interaction scheme for such a program for the test system and the task (discussed in the next section) implemented by OpenMP tools as an example, can be represented as shown in Fig. 3.

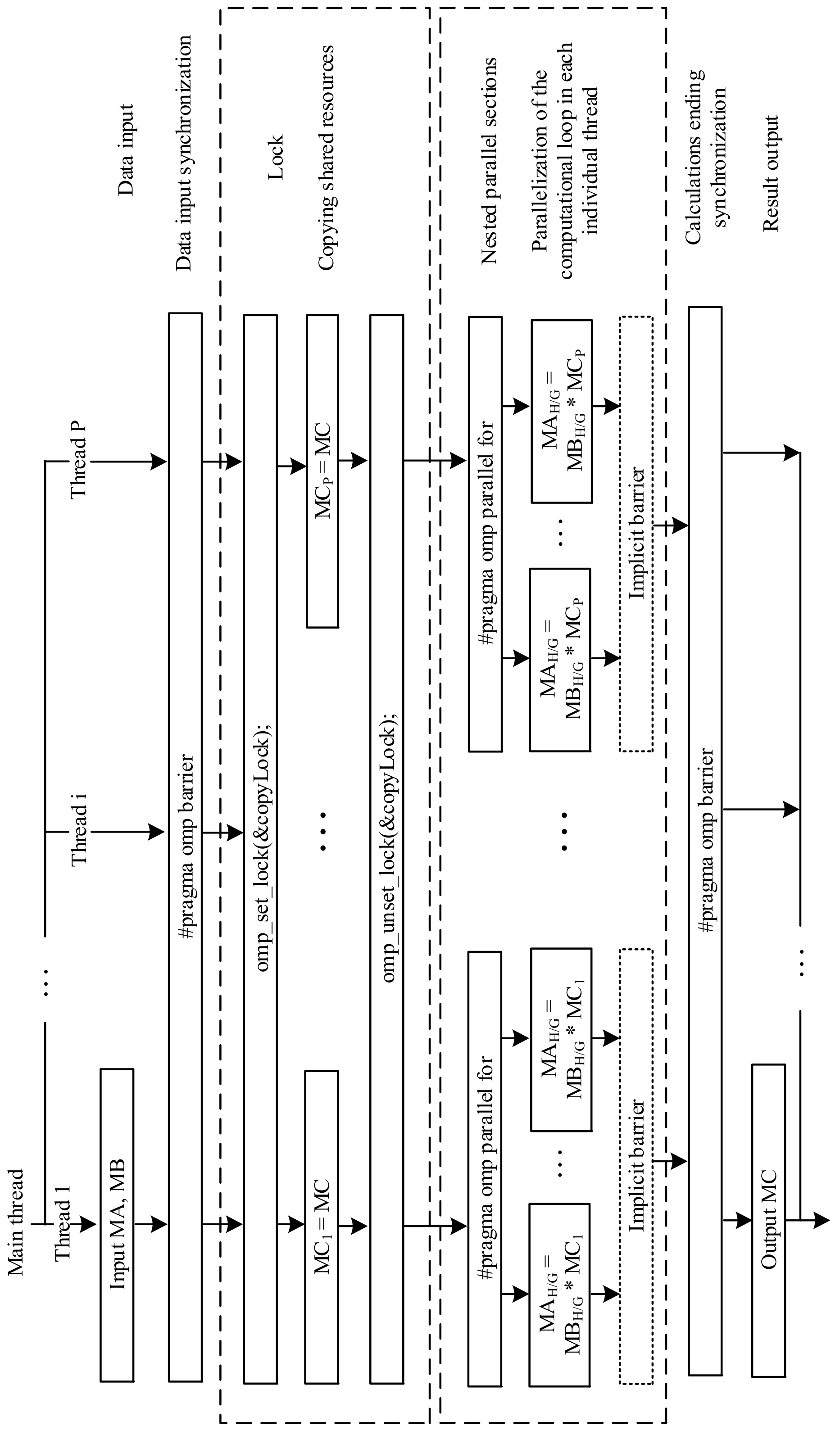

Fig. 3. The scheme of the interaction of threads within the model of nested parallelism implemented by OpenMP tools using the problem of two square (*N***N*) matrices multiplication as an example

Support for nested parallelism in OpenMP is enabled manually. There are two ways to do this:

1. By calling **omp_nested true** command before compiling the program.
2. By calling **omp_set_nested(1)** procedure in the program code.

Both methods set the environment variable ***omp_nested*** to the value of true. However, the second method is more preferable because it provides better portability of the code, because there is no guarantee that on computers where the code will run in the future, will be manually set the appropriate value of the variable *omp_nested*.

Unlike OpenMP, Java and C# do not require any additional operations to activate compiler support for nested parallelism.

EXPERIMENTAL TESTING

Selection a problem for testing

For the most transparent comparative testing, the problem of multiplying two square matrices of large dimension ($N*N$ elements of 64-bit type double) was chosen as an example of a typical problem from the field of high-load computing:

$$MA = MB + MC. \tag{1}$$

Due to the large number of elementary mathematical operations, which can be differently and in different quantities distributed between threads or tasks, the program to solve this problem can be properly implemented using all types of parallelism, while always maintaining high-intensity computations, with minimal downtime for synchronization and data exchange, which is important for more transparent comparative performance testing.

In addition, the choice of this problem for comparative testing of different types of parallelism is also ideal in terms of its coverage of all models of data exchange and interaction between threads or tasks. In medium-grained parallelism, it contains the required fragment of the copy of the shared resource (matrix *MC*) between the threads. In fine-grained parallelism, there is an interaction of tasks without copying shared resources (a direct reference to matrices elements), which is just more typical for applied implementations of this type of parallelism. And in nested parallelism, these two approaches of parallel interaction with shared resources are combined. Therefore, it can be argued that this problem contains all the characteristic cases that occur in solving other common mathematical problems in parallel programs.

Description of the mathematical model of the test problem

The mathematical algorithm for performing this problem (1) is reduced to *N*-fold repetition of the calculation:

$$a_{ij} = \sum_{k=1}^{N} b_{i,k} c_{k,j}\,, \tag{2}$$

where a_{ij}, b_{ij}, c_{ij} are the corresponding elements of the i^{th} row and j^{th} column of the matrices *MA*, *MB*, *MC*; i and j lie in the range from 1 to N.

Therefore, the total number of elementary operations that must be performed to obtain solution (1) is equal to N^3.

With this in mind, the following parallel mathematical algorithm was chosen [15]:

$$MA_H = MB_H + MC\,, \tag{3}$$

where H is a rounded up number equal to the quotient of the division N/P and MA_H, MB_H are the corresponding rectangular matrices of dimensions H by N elements of the matrices MA and MB [15].

It is worth noting that in this case, the variable P can be either equal to the number of cores available in the system (medium-grained parallelism), or be less than this number (no parallelism, some variants of nested parallelism), or significantly larger than it (fine-grained parallelism).

Each thread implements H repetitions of the algorithm (2), while calculating the H^{th} part (3) of the total program's result.

Since this algorithm provides for frequent access to all elements of the MC matrix from each thread, an important point is that when creating threads, each of them gets its own instance of this matrix, which eliminates conflicts between threads for capturing and owning shared resources. However, such copying is valid only for medium-grained parallelism; for fine-grained and nested parallelism, such copying does not occur.

Description of the test software and hardware complex

Testing of programs was carried out in two identical MCSs. Their main characteristics are shown in Table 1.

Table 1. The main characteristics of the test MCSs

Hardware	
Processor	AMD Phenom II
Processor architecture	K10
Number of cores	6
RAM capacity	8 Gb
RAM type	DDR3
Sofrware	
Operation system	Windows 7
OpenMP version	3.1
C++ compiler	MC++ (MSVC)
JVM version	1.8
.NET Framework version	4.7

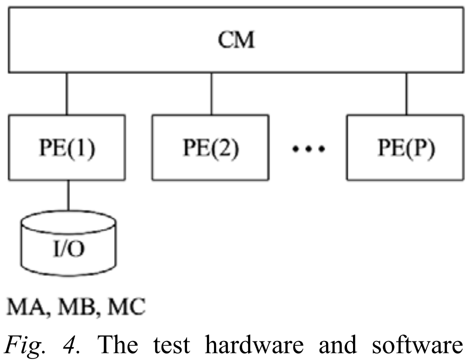

Fig. 4. The test hardware and software complex schematic structure

The software structure is aligned with the hardware structure (primarily for medium-grained part of computing), and it includes both manual and automatic scalability. The schematically formed structure of the hardware and software test complex is shown in Fig. 4.

The following symbols are introduced into this diagram:

1. PE(i) — the processing ele-

ment (processor or core). Its index corresponds to the number of this element, and lies in the range from 1 to P inclusive, where P is the number of all processing elements in the test system. At the software level, each physical processor element corresponds to a software-generated thread.

2. CM — the common (shared) memory (RAM for example), to which each processor element is connected, and with the help of which they communicate with each other.

3. I/O - an I/O device that is connected to one of the processor elements (or to one of the cores, since in fact the processing of I/O signals will ultimately be processed by one of the physical cores of the system), and which provides input of initial data and output of results.

4. MA, MB, and MC are matrices that make up the multiplication operation (1). Moreover, the MA and MB matrices are multipliers and are entered from the I/O device (for large dimensions, random input is used, the execution time of which is not taken into account in the overall test result), and the MC matrix is the result that is output by this device at the end of all calculations (the execution time measurement is stopped just before the output).

Tests results

Table 2 shows the results of testing parallel programs for the operation of matrix multiplication for different values of *N* (dimension of matrices). Presented the execution time of programs that were built with:

- using medium-grained parallelism through the thread mechanism;
- using fine-grained parallelism through the parallel loops and/or fork-join mechanisms described in the previous section;
- using nested (combined) parallelism, when each thread additionally uses parallel processing through the parallel loops and/or fork-join constructs.

Based on the actual time indicators obtained, the acceleration coefficients (speedup coefficients) of the considered programs are calculated. The speedup coefficient of a parallel program is the ratio of the execution time of a program without parallelism on one computing core to the execution time of a similar program with parallelism on P computing cores and shows how much the program execution time is reduced in a parallel system [15]. Calculated using the formula

$$SC = \frac{T_1}{T_p},$$

where T_1 is the actual running time of the program without applying parallelism, and T_p is the actual running time of a similar program using parallelism in P computing cores. Ideally, this coefficient is equal to the number of cores, but in practice, this coefficient is always several tenths less than the number of cores. This is due to the presence of other background processes in the system, which also occupy a certain place in the kernel operation plan.

Below are graphs (Fig. 5) showing the dependence of the speedup coefficients for all three types of parallelism on the values of *N*.

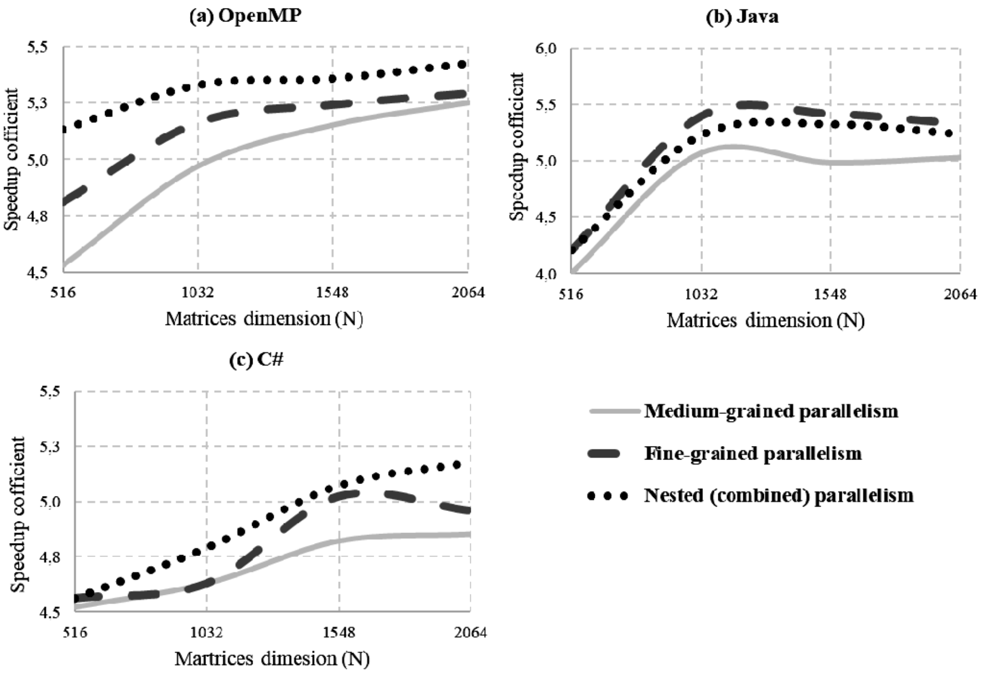

Fig. 5. The dependence of the average speedup coefficients of test parallel programs on the type of parallelism used in them, and on the software tools for its implementation: OpenMP (via C ++) (a), Java (b) and C# (c) in solving the problem of two square matrices multiplication

Additionally, more detailed testing of fine-grained parallelism was conducted in order to identify ways to increase its efficiency. At this stage, those test programs from the developed package were tested, which were written using only fine-grained parallelism. Multiple measurements of the time of the multiplication operation of two matrices with a dimension of 1500*1500 elements were performed. The dependence of execution time on the number of software implemented threads was checked. Fig. 6 demonstrate the detected time dependence on the number of created tasks. More detailed results are provided in Table 2 and 3.

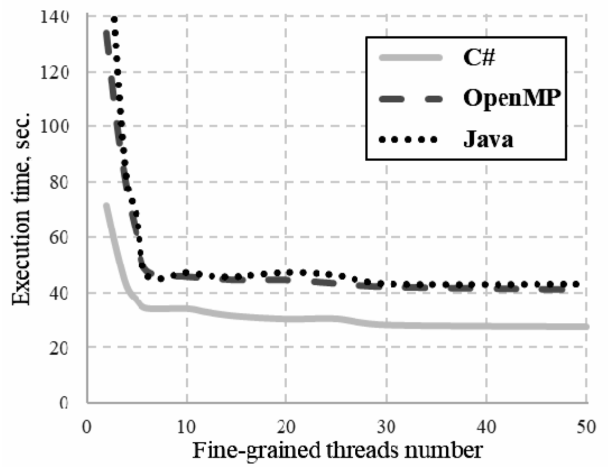

Fig. 6. The dependence of fine-grained parallelism execution time on the number of created threads

Table 2. The results of testing the performance of test parallel programs developed using different types of parallelism (or without using parallelism – as a control sample) and various software tools for its implementation in solving the problem of two square matrices multiplication

N	**Computing time (sec)**					
	Medium-grained			**Fine-grained**		
	OpenMP	Java	C#	OpenMP	Java	C#
516	1,7	0,2	1,2	1,6	0,2	1,1
1032	13,3	3,3	9,8	12,8	2,9	9,8
1548	46,7	14,0	35,6	45,9	12,1	34,1
2064	111,7	38,5	79,8	110,8	33,5	78,1
N	**Computing time (sec)**					
	Nested			**Non-parallel**		
	OpenMP	Java	C#	OpenMP	Java	C#
516	1,5	0,2	1,1	7,7	8,0	5,2
1032	12,4	3,1	9,5	66,1	66,9	45,5
1548	44,9	13,0	33,8	240,6	250,2	171,4
2064	108,1	35,5	74,8	586,5	640,1	387,1

Table 3. The results of fine-grained parallelism testing in solving the problem of two square matrices multiplication using various software tools for its implementation

Number of threads	**Computing time (sec)**		
	C#	Java	OpenMP
2	71,309	133,944	190,618
3	55,1	101,41	125,12
4	42,099	77,393	79,894
5	37,202	60,942	68,167
6	34,222	47,532	46,157
10	34,113	45,551	47,202
12	32,841	45,117	46,011
15	31,405	44,37	45,7
20	30,404	44,33	47,196
25	30,511	43,101	46,21
30	28,329	41,9	43,15
50	27,683	40,88	43,01

CONCLUSIONS AND FUTURE WORK

The results obtained during testing showed the effectiveness of MCS in the implementation of the considered mathematical problem solution by using Java and C# languages and OpenMP library. Additionally, reduction of the programs execution time with the application of the parallelism of any degree of grain is possible (speedup coefficient values are in the range of 4,0–5,5). The best result in terms of program execution time was obtained for the C#.

Medium-grained parallelism showed sufficient efficiency, but had the worst result. At the same time, the speedup achieved by OpenMP tools is constantly increasing with the amount of data processed. While using the C# and the Java tools, speedup remains at approximately the same level.

Using fine-grained parallelism was also effective, in which case the speedup coefficient increases steadily with increasing data volume. This type of parallelism was most effective in C#.

Furthermore, additional testing of fine-grained parallelism revealed a declining exponential dependence between the number of threads, which is allowed to create by the program, and the time of its operation. As mentioned in Section 3, the main reason for this is a larger number of fine-grained tasks (threads) that access physical processing elements (cores) and shared resources, which leads to significant downtime due to the problem of mutual exclusion. The optimal number of fine-grained tasks is in the range of 5 to 20, regardless of how it was organized.

Nested (combined) parallelism showed its effectiveness and allowed to increase speedup coefficient when it is used in C# language and OpenMP library. Moreover, there is an increase of speedup coefficient with increasing amount of processed data, which is one of the most important arguments for the feasibility of this approach in the MCS.

It could be assumed that the efficiency of using nested parallelism will increase with an increasing number of cores in the MCS, where:

1. There will be additional processor resources for its implementation.
2. It is possible to reduce the size of grains.
3. There will be an optimal ratio between the number of streams and subthreads.

Besides, the efficiency of nested parallelism implemented in OpenMP can be improved by more efficient implementation of the powerful sub-threads management system embedded in the library, similar to how it was done in the fork-join model. Since the identified patterns of change in program execution time and speedup coefficients, in general, are preserved for all considered means of organizing parallelism, it can be argued that this approach will be effective regardless of the language or library by which it will be organized.

Therefore, it can be argued that the use of fine-grained and/or combined (nested) parallelism in most cases is an effective approach to the implementation of parallel computing in multi-core computer systems.

Possible directions of work continuation:

1. Testing formed hypotheses in larger and more powerful multi-core computer systems.
2. Testing formed hypotheses on a number of more applied problems.
3. Check the statement about the effectiveness of fine-grained and combined parallelism, regardless of the instruments of its organization, compared with medium-grained parallelism.

From the point of view of the development of high-level instruments of fine-grained parallelism realization in modern parallel programming languages and libraries:

1. Adaptation of existing computational planning methods to the realities of fine-grained and nested parallelism.
2. Adaptation of existing effective processing queue management policies to the realities of fine-grained and nested parallelism.

REFERENCES

1. *ISO 9000:2000 Quality management systems – Fundamentals and vocabulary*. Available: https://www.iso.org/standard/29280.html
2. L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979. doi: 10.1109/TC.1979.1675439.
3. A.C. Wayne, S.J. Procter, and T.E. Anderson, "The nachos instructional operating system", in *USENIX Winter 1993 Con-ference (USENIX Winter 1993 Conference)*. San Diego, CA: USENIX Association, Jan. 1993. doi: 10.1.1.181.878.
4. B.S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. USA: Prentice Hall Press, 2014. doi: 10.5555/2655363.
5. J.E. Moreira, D. Schouten, and C.D. Polychronopoulos, "The performance impact of granularity control and functional parallelism", in *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, *ser. LCPC'95*, pp. 581–597. Berlin, Heidelberg: Springer-Verlag, 1995. doi: 10.5555/645673.665710.
6. K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, 1st ed. McGraw-HillHigher Education, 1992. doi: 10.5555/541880.
7. R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*. Cambridge, Mass: MITPress, 1996. doi: 10.5555/249608.
8. B. Blaise, *Introduction to Parallel Computing*. [Online]. Available: https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial/
9. J.L. Hennessy and D.A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2011. doi: 10.5555/1999263.
10. O. Bandman, "Composing fine-grained parallel algorithms forspatial dynamics simulation", in *Proceedings of the 8th International Conference on Parallel Computing Technologies, ser. PaCT'05*, pp. 99–113. Berlin, Heidelberg: Springer-Verlag, 2005. doi: 10.1007/11535294_9.
11. D. Lea, "A java fork/join framework", in *Proceedings of the ACM2000 Conference on Java Grande, ser. JAVA '00*, pp. 36–43. New York, NY, USA: Association for Computing Machinery, 2000. doi: 10.1145/337449.337465.
12. P.E. Hadjidoukas, G.C. Philos, and V.V. Dimakopoulos, "Exploiting fine-grain thread parallelism on multicore architectures", *Sci. Program.*, vol. 17, no. 4, pp. 309–323, Dec. 2009. doi: 10.1155/2009/249651.
13. P. Czarnul, "Assessment of OpenMP master-slave implementations for selected irregular parallel applications", *Electronics*, vol. 10, no. 10, 2021. doi: 10.3390/electronics10101188.
14. J. Ponge, *Fork and Join: Java Can Excel at Painless Parallel Programming Too!* [Online]. Available: https://www.oracle.com/technical-resources/articles/java/fork-join.html
15. O. Rusanova and A. Korochkin, "Scheduling problems for parallel and distributed systems", *Ada Lett.*, vol. XIX, no. 3, pp. 195–201, Sep. 1999. doi: 10.1145/319295.319323.

Received 15.06.2022

INFORMATION ON THE ARTICLE

Valerii V. Martell, ORCID: 0000-0002-1749-5818, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Ukraine, e-mail: valerii.martell@gmail.com

Aleksandr V. Korochkin, ORCID: 0000-0003-4650-2316, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Ukraine, e-mail: avcora@gmail.com

Olga V. Rusanova, ORCID: 0000-0003-0145-3012, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Ukraine, e-mail: olga.rusanova.v@gmail.com

ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ ДРІБНОЗЕРНИСТОГО ТА ВКЛАДЕНОГО ПАРАЛЕЛІЗМУ ДЛЯ ЗБІЛЬШЕННЯ ПРИШВИДШЕННЯ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ У БАГАТОЯДЕРНИХ КОМП'ЮТЕРНИХ СИСТЕМАХ / В.В. Мартелл, О.В. Корочкін, О.В. Русанова

Анотація. Подано результати порівняльного аналізу ефективності використання паралелізму різного ступеня зернистості в сучасних багатоядерних комп'ютерних системах з використанням найпопулярніших натепер мов програмування та бібліотек (таких як C#, Java, C++ та OpenMP). Досліджено можливості підвищення ефективності обчислень у багатоядерних комп'ютерних системах за допомогою комбінацій середньо- та дрібнозернистого паралелізму. Отримані результати демонструють високу потенційну ефективність використання дрібнозернистого паралелізму для організації інтенсивних паралельних обчислень. На підставі цих результатів можна стверджувати, що порівняно з більш традиційними методами розпаралелювання, які використовують паралелізм із середньою зернистістю, використання окремо дрібнозернистого паралелізму може скоротити час обчислення великої тестової математичної задачі в середньому на 4% , а використання комбінованого паралелізму — до 5,5%. Це скорочення часу виконання доцільне в разі виконання надвеликих обчислень.

Ключові слова: багатоядерна комп'ютерна система, ядро, потік, завдання, паралелізм, зернистість, fork-join, коефіцієнт пришвидшення, дрібнозернистий паралелізм, вкладений паралелізм, комбінований паралелізм.