UDK [611.018.51+615.47]:612.086.2

COOPERATIVE MULTI-THREAD SCHEDULER FOR SOLVING LARGE-SIZE TASKS ON MULTI-CORE SYSTEMS

O.N. Karasik
Tech Lead at ISsoft Solutions (part of Coherent Solutions) in Minsk, Belarus, PhD in Technical Science

A.A. Prihozhy
Professor at the Computer and System Software Department,
Doctor of Technical Sciences,
Full Professor
Belarusian National Technical University

ISsoft Solutions (part of Coherent Solutions), Belarus
Belarusian National Technical University, Belarus
E-mail: karasik.oleg.nikolaevich@gmail.com, prihozhy@yahoo.com

O.N. Karasik

Tech Lead at ISsoft Solutions (part of Coherent Solutions) in Minsk, Belarus; PhD in Technical Science (2019). Interested in parallel computing on multi-core and multi-processor systems.

A.A. Prihozhy

Full professor at the Computer and system software department of Belarusian national technical university, doctor of science (1999) and full professor (2001). His research interests include programming and hardware description languages, parallelizing compilers, and computer aided design techniques and tools for software and hardware at logic, high and system levels, and for incompletely specified logical systems. He has over 300 publications in Eastern and Western Europe, USA and Canada. Such worldwide publishers as IEEE, Springer, Kluwer Academic Publishers, World Scientific and others have published his works.

Abstract. The architecture of a cooperative multi-thread scheduler for thread execution on multi-core systems that run Windows is proposed. The architecture is implemented using the User Mode Scheduling (UMS) mechanism, which allows the user application to organize cooperative thread execution. The architecture under development includes the necessary set of components: a user thread for executing user code; new synchronization primitive for organizing the interaction of user threads running on different cores of a multicore system; a control transfer mechanism between user threads running on the same core. The architecture allows the programmer to implement cooperative multi-threaded algorithms to accelerate the solution of large-scale problems on multi-core systems.

Keywords: multi-threaded application, multi-core system, cooperative multi-tasking, scheduler, large-size tasks.

Introduction. Solving large-scale problems in reasonable time is imposible without exploring parallelism of modern parallel systems. Effectiveness of the parallelization depends on the existence of a parallel solution for the problem, on the possibility to develop an effective parallel algorithm for the problem, and on the availability of hardware and software tools, which meet the requirements of the parallel algorithm under implementation [2]. Nowadays, a wide range of tools for the development of parallel applications exist: Windows API, OpenMP, Cilk Plus, Portable Operating System Interface Threads (POSIX Threads or PThreads), Threading Building Blocks (TBB), Open Computing Language (OpenCL), different implementations of the Message Passing Interface (MPI), and others [2-8]. Choosing the appropriate tool depends on various aspects, including the operating system compatibility, the availability (licensing, freeware or shareware), and the specifics of the parallel problem. In this paper, we investigate the effectiveness of the cooperative multi-thread scheduler implementations [9] on two different parallel applications: the Gaussian Elimination algorithms [10, 11] for solving algebraic equations, and the parallel Floyd-Warshall algorithms for all-pairs shortest paths problem [11 – 14].

Cooperative multi-tasking in Windows. All modern versions of the Windows operating system use a preemptive scheduler as the default thread scheduler [15]. However, starting from Windows 7 (for workstations) and Windows Server 2008 R2 (for servers), Windows provides a User-Mode Scheduling (UMS) interface, which allows the execution of an application by operating system scheduler in such a way as to construct a custom thread scheduler [16]. UMS interface consists of three core entities: a worker thread – *UmsWorkerThread* (UMSWT), a scheduler thread - *UmsSchedulerThread* (UMSST), a signature of the scheduling procedure, which is invoked by operating system in certain circumstances (described below) – *UmsSchedulerProcedure* (USP) and a completion list of worker threads– *UmsCompletionList* (UMSCL).

UmsSchedulerThread is responsible for execution of UMSWT. It is implemented using standard operating system thread (executed by operating system scheduler), which is switched to scheduling mode by the invocation of an *EnterUmsSchedulingMode* procedure. The parameters of this procedure are a pointer to UMSCL and a pointer to an implementation of USP. The operating system invokes the provided implementation of USP in the cases as follows: at the initialization of UMSST; immediately after the call to *EnterUmsSchedulingMode* (using the *UmsSchedulerStartup* event); after blocking UMSWT at the system call (*UmsSchedulerThreadBlocked* event); when UMSWT passes the control to UMSST (*UmsSchedulerThreadYield* event). USP is also responsible for handling operating system callbacks, and for maintaining a list of all UMSWT created by the application. UMSST executes UMSWT using the *ExecuteUmsThread* procedure.

UmsWorkerThread is responsible for the execution of a user code. It is implemented with the standard operating system thread, which is transformed into UMSWT by setting up a set of attributes at the moment of creating. These attributes include pre-allocated *UmsContext* (used by the operating system and created using the *CreateUmsThreadContext* procedure), and a pointer to UMSCL. At the initialization, operating system pushes UMSWT to UMSCL previously specified as attribute. Starting from this moment UMSWT is under the control of UMSST bound to UMSCL (cooperative multi-threading). UMSWT can pass the control to UMSST using the *UmsThreadYield* procedure.

Figure 1 illustrates the execution of two operating system threads T_1 and T_2 under the control of operating system scheduler, the execution of one user-mode scheduling thread $UMSST_1$, and the execution of two user-mode scheduling worker threads $UMSWT_1$ и $UMSWT_2$.

Architecture of cooperative multi-thread scheduler. The cooperative multi-thread scheduler consists of three core components: scheduler thread (CST), user thread (CUT) and synchronization primitive (CSP).

The scheduler implements:

–the memory management, in particular, aligned memory allocation, NUMA aware memory allocation, and memory buffering;

–procedures for creating / terminating CST, CUT and CSP as well as procedures for interaction between these core components.

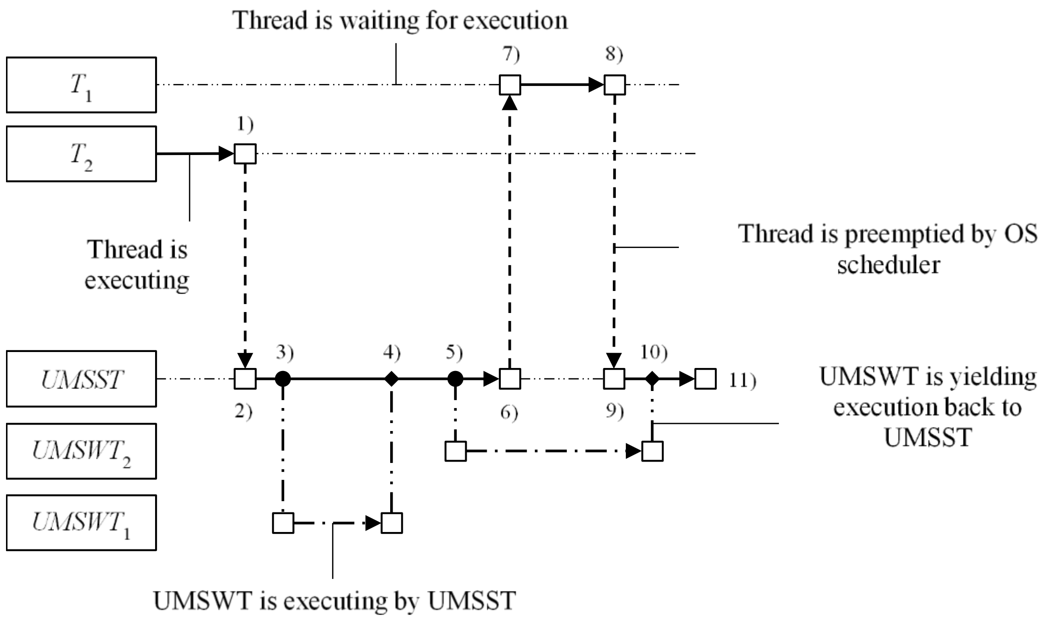

Thread is waiting for execution
T_1
T_2
1)
7)
8)
Thread is executing
Thread is preempted by OS scheduler
UMSST
$UMSWT_2$
$UMSWT_1$
2)
3)
4)
5)
6)
9)
10)
11)
UMSWT is yielding execution back to UMSST
UMSWT is executing by UMSST

Figure *1.* – Illustration of the execution of two operating system threads T_1 and T_2 (under control of the operating system scheduler), one user-mode scheduler thread $UMSST_1$, and two user-mode scheduling worker threads $UMSWT_1$ и $UMSWT_2$: solid arrows represent the execution of operating system threads; long dash dot arrows represent the execution of user-mode scheduler worker threads; dash arrows represent the thread preemption done by the preemptive scheduler; square and diamond glyphs represent the begin and end of the user-mode scheduler thread execution

During the initialization, the scheduler allocates the following resource on each logical processor of the multi-core system:

- one UMSCL;
- one ready user thread queue (RUTQ);
- one CST which is bound to previously allocated UMSCL and RUTQ.

Besides the mentioned above, the scheduler also includes arrays of all created CST, CUT and CSP. A high-level view of cooperative multi-threaded scheduler is illustrated on Figure 2.

Cooperative scheduler thread. CST is responsible for handling the operating system callbacks and the user thread requests. It consists of UMSST, an implementation of *UmsSchedulerProc*, a pointer to RUTQ and a field representing its CST state. CST is created during scheduler initialization. During the initialization, CST switches from *Created* to *Initializing*, to *Initialized* and then to *Executing* state (executes an implementation of *UmsSchedulerProc*). Starting from this point, CST can switch between the four states as follows: *Executing*, *WaitingWrIdle* (if no requests to handle, scheduler thread is idling), *WaitingTaskExecuting* (CST is executing user thread), and *Terminated*. CST supports two type of CUT requests: direct control transfer between two CUT, and blocking-unblocking CUT by using CSP.

A high-level view of the control direct transfer implementation is presented on Figure 3. It is implemented as the following sequence of events:

1. CUT_1 transfers execution back to CST using the *UmsThreadYield* procedure that passes information about CUT_2 over the *SchedulerParam* argument (Figure 3, arc 1);
2. CST uses information from *SchedulerParam* and finds requested CUT_2;
3. CST uses the *ExecuteUmsThread* procedure to execute CUT_2 (Figure 3, arc 2).

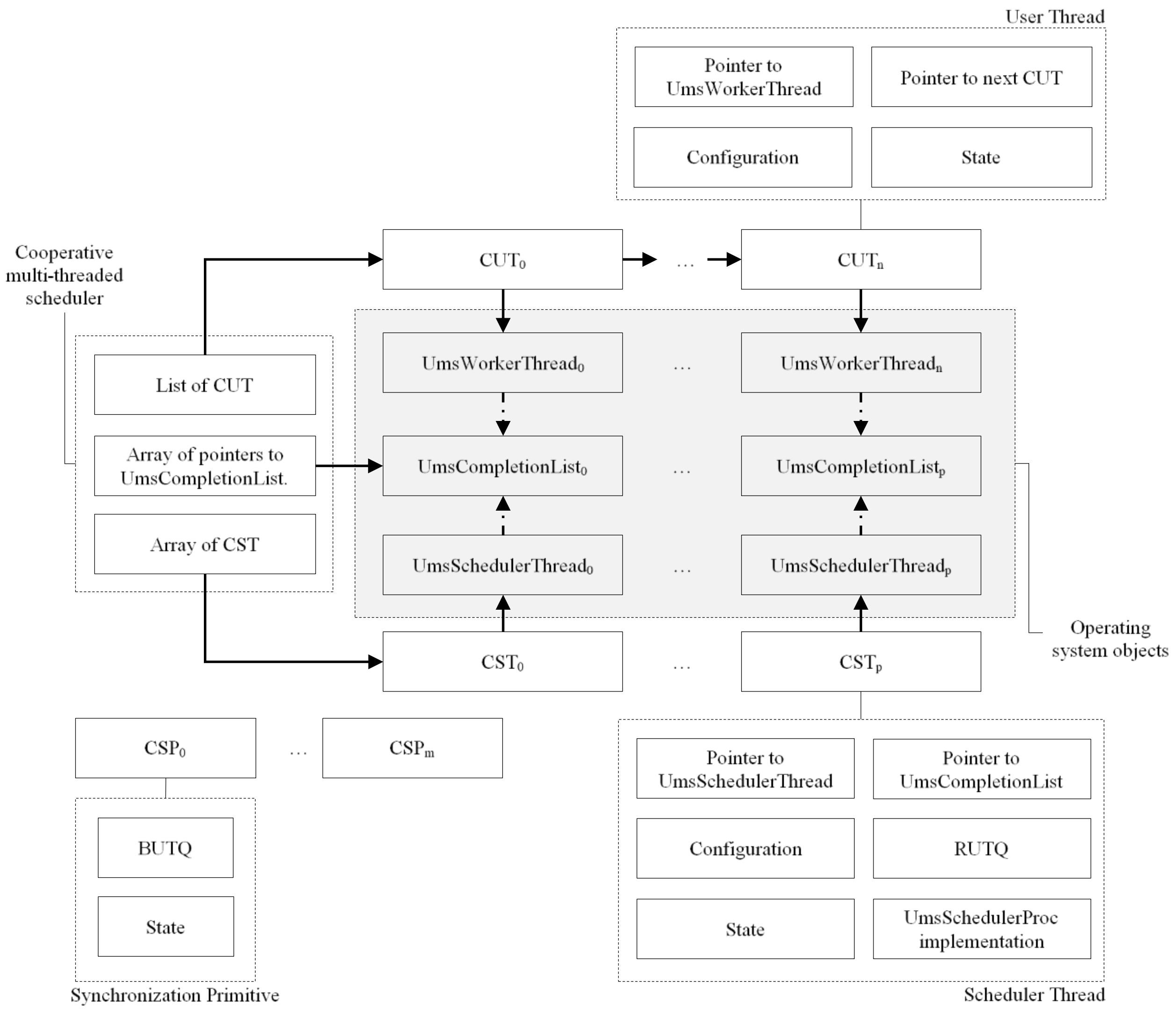


Figure *2.* – High-level view of cooperative multi-thread scheduler architecture

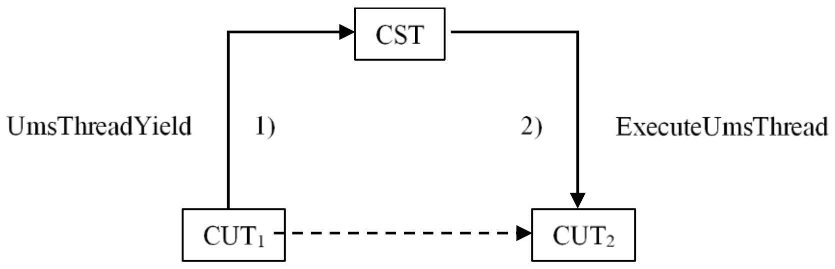


Figure *3.* – High-level view of control direct transfer between CUT_1 and CUT_2

The implementation of CUT on CSP synchronization request handler has been described in [9].

Cooperative user thread. CUT is responsible for the user thread execution. It consists of UMSWT and a field representing the CUT state. CUT is created by a user request which includes a pointer to the user defined procedure and an index of the logical core which CUT will execute on. During the initialization, CUT switches from the *Created* state to *Initializing*, then to *Initialized* and then to *Ready* states (CUT is ready for execution and currently resides in RUTQ). Starting from this moment, CUT could switch between these five states: *Ready*, *Executing*, *WaitingWrStopped* (CUT is blocked after performing the control transfer), *WaitingWrBlocked* (CUT is synchronized using CSP), *WaitingWrSystemTrap* or *WaitingWrSystemCall* (CUT is blocked on a system call) and *Terminated*. When CUT is switching between the above states it sometimes holds one of the intermediate states: *StandBy* (when CUT has been selected from RUTQ but has not been executed yet, i.e. between *Ready* and *Executing*), and *WaitingWrYielded* (when CUT has transferred execution, but the request has not been decoded yet by CST, i.e. between *Executing* and *WaitingWrStopped*, or between *Executing WaitingWrBlocked*).

Cooperative synchronization primitive. CSP represents an event. It is used to synchronize two or more CUTs executing on different cores. CSP allows to block CUT waiting for the event, and it resumes CUT's execution without support from the operating system. CSP is created by a user request and is not tied to any logical core, CST or CUT. CSP consists of a blocked user thread queue (BUTQ), which is used to hold blocked CUTs, and includes a field for representing CSP state.

At any moment of time, CSP can be in one of the following states: *Reset*, *Signaled*, *Signaling*, *SignalingActSignal*, *SignalingActReset*, *Joining*, *JoiningActSignal* and *JoiningActReset*. States *Reset* and *Signaled* are terminal states. Other states are transitional ones, which are required to support concurrent calls to CSP's procedures.

In order to better understand the role of each state, let us consider five examples.

Example 1. CSP is in state *Reset*. It indicates that the event represented by CSP has not happened yet. CUT_1 wants to notify about the event all the CUTs blocked on CSP. The notification should either unblock one of the CUTs or switch CSP into the *Signaled* state. Here is the sequence of events:

Step 1. CUT_1 switches CSP from state *Reset* to state *Signaling*. This is to indicate that there is a CUT which is executing the notify procedure.

Step 2. CUT_1 checks BUTQ for the blocked CUT. If BUTQ does not contain the blocked CUT then the behavior of CUT_1 is described by step 3a, otherwise by step 3b.

Step 3*a*. Because BUTQ does not contain a blocked CUT, CUT_1 switches CSP from state *Signaling* to state *Signaled*. It indicates that the event represented by CSP has happened.

Step 3*b*. Because BUTQ contains one or more blocked CUTs, CUT_1 pops one blocked CUT and moves it from BUTQ to RUTQ (simultaneously switching CUT to state *Ready*). Then CSP switches to state *Reset*.

Example 2. Imagine, that at the same time as CUT_1 executes the "signal" action, one more CUT_2 also want to execute the "signal" action to notify about the event. Here is how the sequence of events will look like:

Step 1. CUT_2 fails to switch CSP from state *Reset* to state *Signaling*. This is because CSP is already in state *Signaling*.

Step 2. Because CUT_2 cannot be sure whether CUT_1 has already checked for BUTQ, it switches CSP from state *Signaling* to state *SignalingActSignal*. This state indicates that if CUT_1 have not unblocked any of CUTs, then it should switch CSP back to state *Signaling* and repeat the check of BUTQ.

Example 3. Imagine, CUT_2 from the previous example executes the "reset" action rather than the "signal" one. In this case, the "reset" action will have the same sequence of events as the "signal" action has (see the previous example), with one exception, CUT_2 will switch CSP to state *SignalingActReset*. This will force CUT_1 to switch CSP to state *Reset* right after completion of the "signal" action.

The blocking of CUT is performed in a similar way, with the only exception that the blocking operation is performed by CST. The following two examples illustrate it.

Example 4. CUT_3 transfers control to CST_1 in order to synchronize itself on CSP, which is in state *Reset*. Here is how the execution of the "synchronize" action will look like:

Step 1. CST_1 switches CSP from *Reset* to *Joining* state. It indicates that CSP is now used by CST to synchronize CUT.

Step 2. CST_1 puts CUT_3 in BUTQ.

Step 3. CST_1 switches CSP back to state *Reset*.

Example 5. Imagine, that at the same time as CUT_3 is being synchronized on CSP, other CUT_1 executes the “signal” action. In this case, the “signal” action execution is as follows:

Step 1. CUT_1 fails to switch CSP from state *Reset* to state *Signaling*, as CSP is already in state *Joining*.

Step 2. CUT_1 switches CSP from state *Joining* to state *JoiningActSignal*. This state instructs CST_1 to switch CSP to state *Signaling* and to re-execute the “signal” action after the completion of the “synchronize” action.

Figure 4 shows the state transfer matrix of CSP.

	R	S	SG	SR	SS	JG	JR	JS
R			f_s			f_j		
S	f_r		f_s					
SG		f_s		f_s, f_{cr}	f_s, f_{cs}			
SR	-							
SS			-					
JG	-						f_j, f_{cr}	f_j, f_{cs}
JR	-							
JS			-					

State abbreviations:

S	– Signaled
R	– Reset
SG	– Signaling
SR	– SignalingActReset
SS	– SignalingActSignal
J	– Joining
JR	– JoiningActReset
JS	– JoiningActSignal

Conditional flags:

f_r	– flag, indicating a reset operation was taken
f_s	– flag, indicating a signal operation was taken
f_j	– flag, indicating a join operation was taken
f_{cr}	– flag, indicating a concurrent reset operation was taken
f_{cs}	– flag, indicating a concurrent signal operation was taken
f_{cj}	– flag, indicating a concurrent join operation was taken

Figure *4*. – State transfer matrix of cooperative synchronization primitive

Experimental environment. All experiments were done on two multi-core systems. The first multi-core system was equipped with two Intel Xeon E5520 processors. Each processor has 4 cores. Each core runs on 2.26 GHz frequency and has high-speed hierarchical cache memory (L1 – 64 KB, L2 – 256 KB). Besides that, each core has the Intel Hyper-Threading technology built in, which allows the execution of two hardware threads on the single core. Each processor has access to shared L3 cache of 8 MB size, and access to local and remote memory with NUMA memory organization. The system is equipped with 16 GB of RAM and is controlled by Windows Server 2012 R2 (64 bits). The second multi-core system is equipped by one Intel Core i5-3450 processor (4 cores). Each core runs on 3.10 GHz frequency and has access to the high-speed local hierarchical cache memory (L1 – 256 KB and L2 - 1 MB), and the shared L3 cache memory with capacity of 6 MB. The system is equipped with 16 GB RAM and is controlled by Windows 10 Professional 1809.

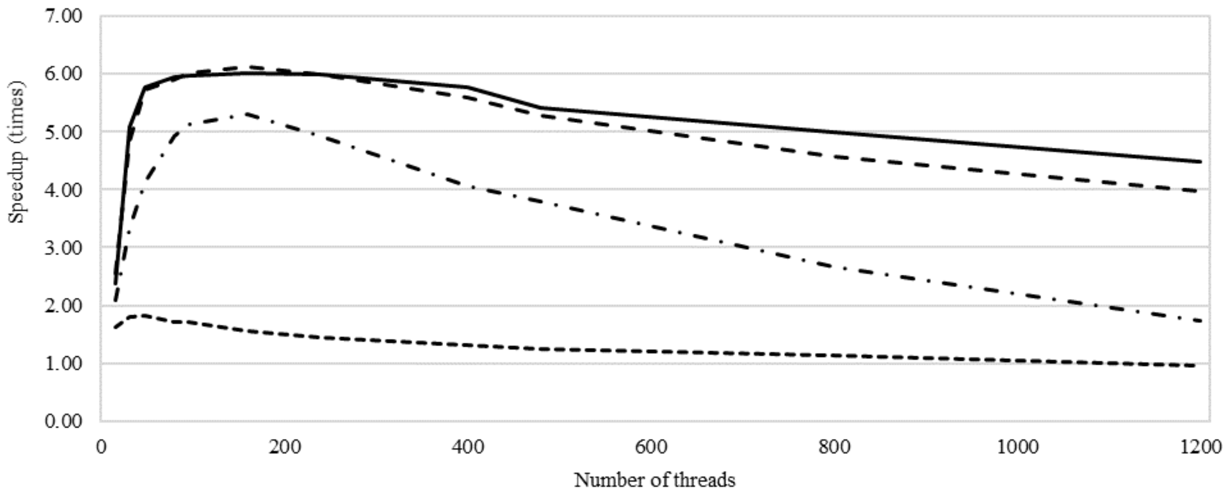
Cooperative scheduler and algorithms implementation. Cooperative multi-thread scheduler is implemented in C/C++ language as dynamically linked library (.dll) using Visual C++ 14.1 compiler.

Parallel Gaussian Elimination algorithms (μ1, μ2) [13, 17] are implemented using the native Windows threads and the *AutoResetEvent* synchronization primitive. Cooperative algorithms (μ1κ и μ2κ) [10, 11] are implemented using the developed scheduler library. The source code of μ1, μ2, μ1κ and μ2κ algorithms are written in C/C++ language and compiled into executions (.exe) using Visual C++ 14.1 compiler. Block-parallel all pairs shortest path Floyd-Warshall algorithm (BFW) is implemented using OpenMP directives for the task-based parallelism. The cooperative threaded block-parallel algorithm CTBPA [12] is implemented using the scheduler library. The source code of both algorithms is written in C/C++ language and is compiled using Intel Compiler 18 that is configured on maximum optimization (OV3) with additional options to do maximum optimization for underlying multi-core system (IvyBridge specific optimization and vectorization using Intel AVX).

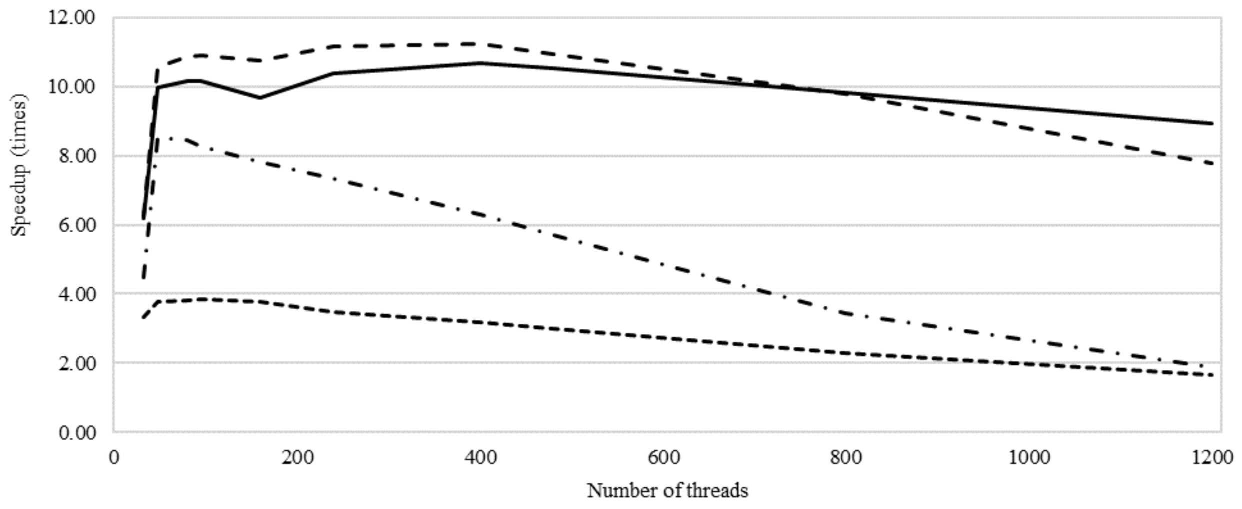
Experimental results. In order to demonstrate the effectiveness of the cooperative multi-thread scheduler and the developed cooperative algorithms, we conducted experiments using two different parallel implementations of the Gaussian Elimination and the block-parallel implementation of all-pairs shortest path Floyd-Warshall algorithm. Figures 5, *a* and 5, *b* report experimental results obtained in series of 1000 runs for each of four μ1, μ2, μ1κ and μ2κ algorithms depending on the number of threads. All experiments use linear algebraic equations of 2400 variables. The obtained results are compared against the results obtained for single-threaded implementations on both multi-core systems. The best execution time of the single-threaded implementation is 10.43 sec on the first multi-core system and is 4.91 sec on the second multi-core system. On first multi-core system, the cooperative algorithms μ1κ and μ2κ have shown a maximum speed up of 6.00 and 6.12 times, which exceeds the maximum speedup shown by the μ1 and μ2 algorithms (5.30 and 1.82 times respectively). On the second multi-core system, cooperative algorithms μ1κ and μ2κ have shown the maximum speed up of 10.68 and 11.23 times, which significantly overcomes the speed up obtained by the μ1 and μ2 algorithms (8.47 and 3.83 times respectively).

In addition to the execution time of four algorithms μ1, μ2, μ1κ and μ2κ, we have analyzed the execution time distribution in series of 1000 runs of each algorithm. Figure 6 presents histograms of the execution time and time intervals (in sec) for both multi-core systems. Table 1 reports the execution time intervals of best runs of the algorithms.

Figure 7 presents the execution speedup of algorithm CTBPA comparing to BFW algorithm depending on block size for graph of 2400 vertexes in series of 1000 runs done on Intel Core i5-3450 multi-core system. CTBPA demonstrated speedup for all block sizes with maximum speedup in 26.8% (0.842 vs. 0.664 seconds) for block size of 120x120. Histograms of execution intervals of both algorithms are presented on picture 8. Table 2 demonstrates significant advantage of developed (using developed cooperative multi-threading scheduler) CTBPA algorithm against existing parallel implementation of BFW, done using operating system scheduler.



a)



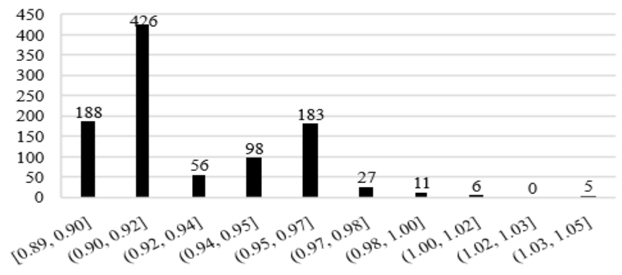
b)

a) – Intel Core i5-3450; b) – Intel Xeon E5520

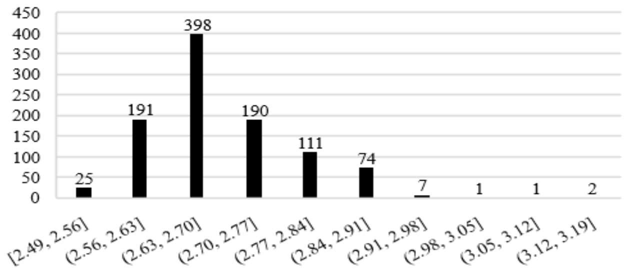
Figure *5.* – Speedup in times between execution time of µ1к (solid), µ2к (dash), µ1 (dash dot), µ2 (dot) algorithms and best execution time of single-threaded algorithm in series of 1000 runs depending on number of threads on SLAE of 2400 variables size

Table 1. – Execution time intervals of µ1, µ2, µ1к и µ2к algorithms for both experimental multi-core systems

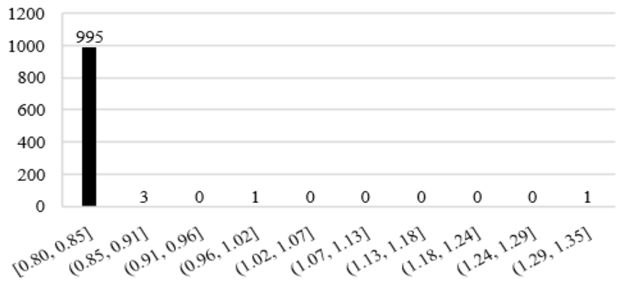
Algorithm	Intel Core i5-3450, 4 cores			2 x Intel Xeon E5520, 8 cores		
	from (seconds)	to (seconds)	width (seconds)	from (seconds)	to (seconds)	length (seconds)
µ1 (160 threads)	0.89	1.05	0.16	1.09	1.85	0.76
µ2 (48 threads)	2.49	3.19	0.70	2.61	2.95	0.34
µ1к (160 threads)	0.80	1.35	0.55	0.969	1.005	0.044
µ2к (160 threads)	0.79	0.99	0.20	0.912	0.952	0.040



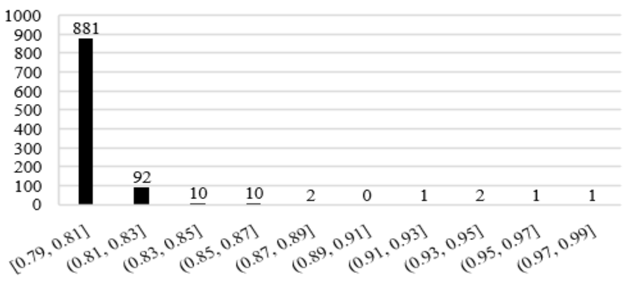
a)



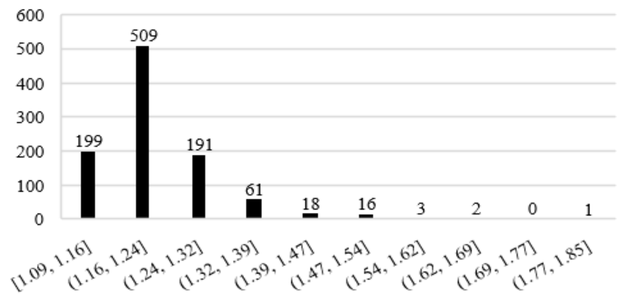
b)



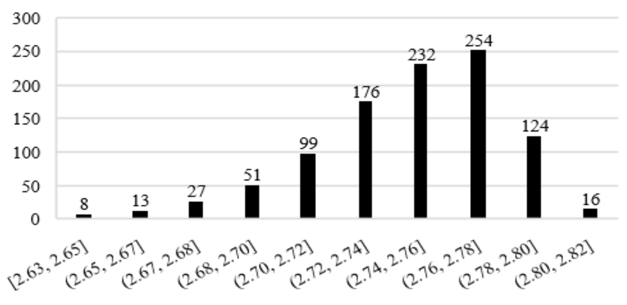
c)



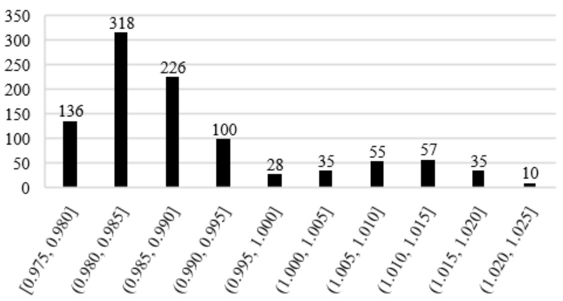
d)



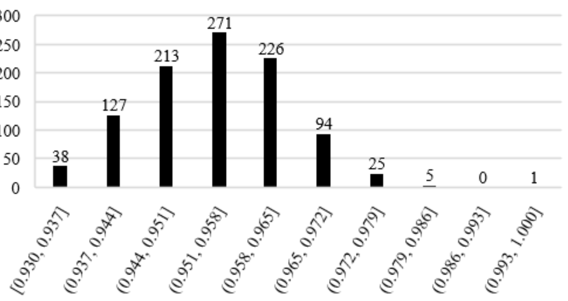
e)



f)



g)



h)

Figure 6. – Histograms of execution time intervals (in seconds) of µ1, µ2, µ1κ and µ2κ algorithms in series of 1000 runs for SLAE of 2400 variables size on Intel Core i5-3450 (4 cores) a), b), c), d)

and Intel Xeon E5520 (8 cores) e), f), g), h)

a) – µ1 (160 threads); b) – µ2 (48 threads); c) – µ1κ (160 threads); d) – µ2κ (160 threads);
e) – µ1 (48 threads); f) – µ2 (96 threads); g) – µ1κ (400 threads); h) – µ2κ (400 threads)

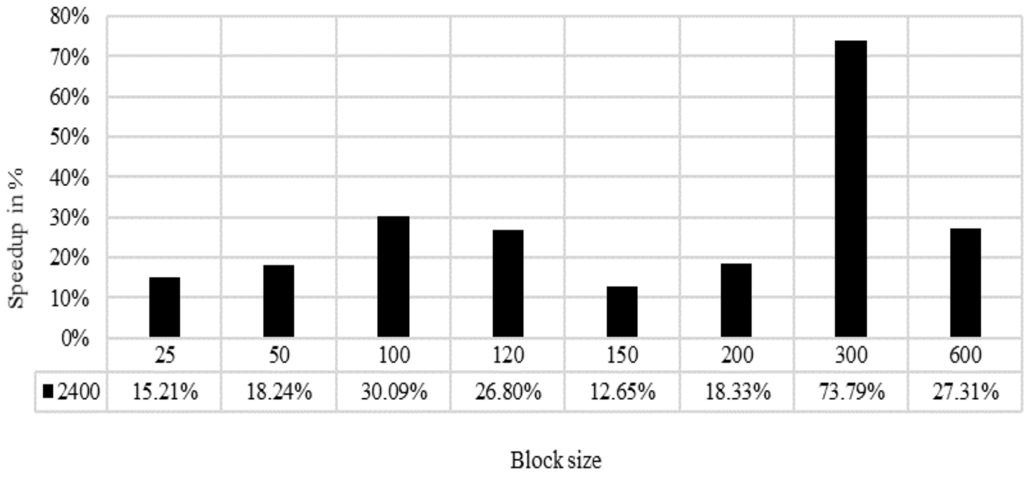
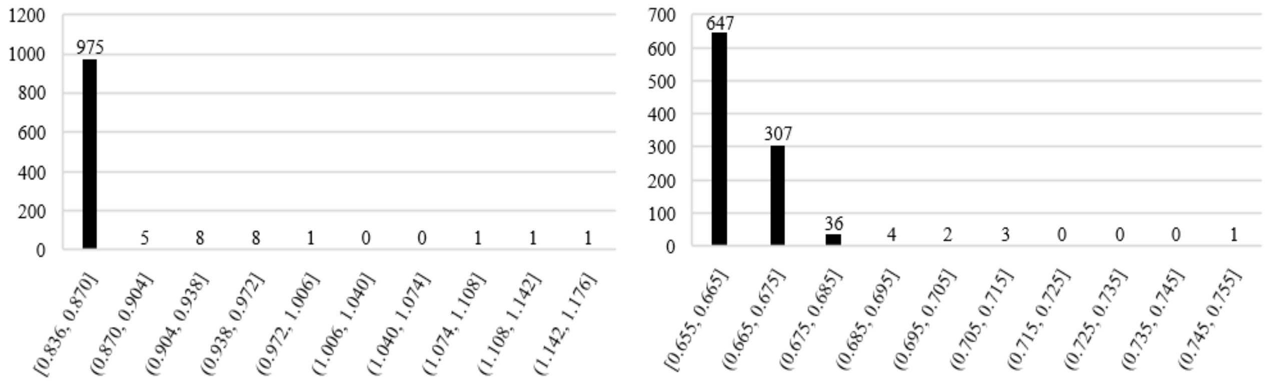


Figure *7.* – Speedup in % given by algorithm CTBPA comparing to algorithm BFW vs. block size on graphs of 2400 vertices in series of 1000 runs done on Intel Core i5-3450



a) b)

Figure *8.* – Histograms of the execution time (sec) of *a*) BFW and *b*) CTBPA algorithms in series of 1000 runs on graphs of 2400 vertices and block size of 120×120

Table 2. – Execution time intervals given by algorithms BFW and CTBPA on block size 120x120

Intel Core i5-3450, 4 core			
Algorithm	from (sec)	to (sec)	width (sec)
BFW	0.836	1.176	0.340
CTBPA	0.655	0.755	0.100

Conclusion When solving a large task, the performance of a multi-core system depends heavily on the operating system and the system software built into it, on the one hand, and depends on the method of constructing parallel multi-threaded algorithms that solve the application problems and perform calculations on a large amount of data, on the other hand. In the paper, a cooperative multi-thread scheduler is proposed and experimentally investigated, which reduces the execution time compared to the preemptive scheduler of the operating system, and is used to create multi-threaded algorithms in such a way that the threads running on one core directly transfer control to each other, and the threads running on different cores interact with each other through the proposed synchronization primitive, that speeds up the processes of thread lock-unlock.

References

[1.]Prihozhy, A.A. Analysis, transformation and optimization for high performance parallel computing / A.A. Prihozhy / Minsk: BNTU, 2019. – 229 p.

[2.]Richter, J. M. Windows via C/C++ / J. M. Richter, C. Nasarre. – 5th ed. – Microsoft Press, 2007. – 848 p.

[3.]OpenMP. OpenMP [Электронный ресурс]. – Режим доступа: https://www.openmp.org. – Дата доступа: 24.02.2020.

[4.]ISO/IEC/IEEE 9945:2009 Information technology — Portable Operating System Interface (POSIX®) Base Specifications, Issue 7 [Электронный ресурс]. – Режим доступа: https://www.iso.org/standard/50516.html. – Дата доступа: 24.02.2020

[5.]MPI Forum. MPI Forum [Электронный ресурс]. – Режим доступа: www.mpi-forum.org. – Дата доступа: 24.02.2020.

[6.]Google. The Go Programming Language [Электронный ресурс]. – Режим доступа: https://golang.org. – Дата доступа: 24.02.2020.

[7.]The Khronos Group Inc. OpenCL [Электронный ресурс]. – Режим доступа: https://www.khronos.org/opencl/. – Дата доступа: 24.02.2020.

[8.]Berkeley Lab. Berkeley UPC - Unified Parallel C [Электронный ресурс]. – Режим доступа: http://upc.lbl.gov/. – Дата доступа: 24.02.2020.

[9.]Карасик, О. Н. Усовершенствованный планировщик кооперативного выполнения потоков на многоядерной системе / О. Н. Карасик, А. А. Прихожий // Системный анализ и прикладная математика. – 2017. – № 1. – С. 4–11.

[10.] Прыхожы, А. А. Кааператыўныя блочна-паралельныя алгарытмы рашэння задач на шмат'ядравых сістэмах / А. А. Прыхожы, А. М. Карасік // Сістэмны аналіз і прыкладная інфарматыка. – 2015. – № 2. – С. 10–18.

[11.] Прихожий, А.А. Кооперативная модель оптимизации выполнения потоков на многоядерной системе / А.А. Прихожий, О.Н. Карасик // Системный анализ и прикладная информатика, 2014, № 4, с. 13-20.

[12.] Карасик, О. Н. Потоковый блочно-параллельный алгоритм поиска кратчайших путей на графе / О. Н. Карасик, А. А. Прихожий // Доклады БГУИР. – 2018. – № 2. – С. 77–84.

[13.] Прихожий, А.А. Исследование методов реализации многопоточных приложений на многоядерных системах / А.А. Прихожий, О.Н. Карасик // Информатизация образования, 2014, № 1, с. 43-62.

[14.] Прихожий, А. А. Кооперативная потоковая модель решения задач большой размерности на многоядерных системах / А. А. Прихожий, О. Н. Карасик // Big Data and Advanced Analytics. – 2018. – №. 4. – С. 381–386.

[15.] Probert, D. Dave Probert: Inside Windows 7 - User Mode Scheduler (UMS) [Электронный ресурс]. / D. Probert. – Режим доступа: https://channel9.msdn.com/shows/Going+Deep/Dave-Probert-Inside-Windows-7-User-Mode-Scheduler-UMS/. – Дата доступа: 01.11.2019.

[16.] Microsoft. User-Mode Scheduling [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/en-us/windows/desktop/ProcThread/user-mode-scheduling. – Дата доступа: 24.02.2020.

[17.] Howe, J. Parallel Gaussian Elimination [Электронный ресурс]. / J. Howe, S. Bratcher. – Режим доступа: http://www.cse.ucsd.edu/classes/fa98/cse164b/Projects/PastProjects/LU. – Дата доступа: 01.11.2018

[18.] Venkataraman, G. Blocked All-Pairs Shortest Paths Algorithm / G. Venkataraman, S. Sahni, S. Mukhopadhyaya // Journal of Experimental Algorithmics (JEA). – 2003. – Vol. 8. – P. 857–874

[19.] Tang, P. Rapid development of parallel blocked all-pairs shortest paths code for multi-core computers / P. Tang // IEEE SOUTHEASTCON 2014. – Lexington, KY, USA: IEEE, 2014. – P. 1–7

[20.] Park, J. Optimizing graph algorithms for improved cache performance / J. Park, M. Penner, V. K. Prasanna // IEEE Transactions on Parallel and Distributed Systems. – 2004. – Vol. 15, №. 9. – P. 769–782