Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 1053–1058
DOI: 10.15439/2023F5683
ISSN 2300-5963 ACSIS, Vol. 35

Performance Analysis of a 3D Elliptic Solver on Intel Xeon Computer System

Ivan Lirkov
0000-0002-5870-2588
Institute of Information and Communication Technologies
Bulgarian Academy of Sciences
Acad. G. Bonchev, bl. 25A
1113 Sofia, Bulgaria
ivan.lirkov@iict.bas.bg
http://parallel.bas.bg/~ivan/

Marcin Paprzycki, Maria Ganzha
0000-0002-8069-2152, 0000-0001-7714-4844
Systems Research Institute
Polish Academy of Sciences
ul. Newelska 6
01-447 Warsaw, Poland,
paprzyck@ibspan.waw.pl, maria.ganzha@ibspan.waw.pl
http://www.ibspan.waw.pl/~paprzyck/,
http://pages.mini.pw.edu.pl/~ganzham/www/

***Abstract*—It was shown that block-circulant preconditioners, applied to a conjugate gradient method, used to solve structured sparse linear systems, arising from 2D or 3D elliptic problems, have very good numerical properties and a potential for good parallel efficiency. In this contribution, hybrid parallelization based on MPI and OpenMP standards is experimentally investigated. Specifically, the aim of this work is to analyze parallel performance of the implemented algorithms on a supercomputer consisting of Intel Xeon processors and Intel Xeon Phi coprocessors. While obtained results confirm the positive outlook of the proposed approach, important open issues are also identified.**

I. Introduction

IN THIS contribution, we are concerned with the numerical solution of linear boundary value problems of an elliptic type. After discretization, such problems are reduced to finding the solution of a linear systems in the form $A\mathbf{x} = \mathbf{b}$. In what follows, symmetric and positive definite problems are considered. Moreover, it is assumed that A is a large matrix. Obviously, the term "large" is relative, as what was large in the past, is no longer large. Therefore, it is assumed that the size of the linear system (matrix) is defined as large, in the context of capabilities of currently existing computers.

In practice, large problems of this class are often solved by iterative methods, such as the conjugate gradient (CG) method. At each step of such methods, a single product of A with a given vector $\mathbf{v}$ is needed. Therefore, to minimize number of arithmetic operations, the sparsity of the matrix A should be explored. On the other hand, exploration of sparsity may be in conflict with parallelization (for large number of processors and cores) of the iterative process.

Typically, the rate of convergence of CG methods depends on the condition number $\kappa(A)$ of the coefficient matrix A. Specifically, the smaller $\kappa(A)$ is, the faster the convergence. Unfortunately, for elliptic problems of second order, usually, $\kappa(A) = \mathcal{O}(n^2)$, where n is the number of mesh points in each coordinate direction. Hence, conditioning of the matrix grows rapidly (gets worse) with n. To accelerate the convergence of the iterative process, a preconditioner M is applied within the CG algorithm. The theory of the Preconditioned CG (PCG) methods says that M is a good preconditioner if it significantly reduces the condition number $\kappa(M^{-1}A)$ and, at the same time, if it allows one to efficiently compute the product $M^{-1}\mathbf{v}$, for a given vector $\mathbf{v}$. The third important aspect should be considered, namely, the need for efficient implementation of the PCG algorithm on modern parallel computer systems, see e.g. [1], [2]. Here, again, the question can be raised, what does it mean "modern" as the practical meaning of this term evolves. Establishing how the problem should be approached on a computer current to the time of conducted research is one of the issues that inspired this work.

II. The 3D elliptic problem

Let us now consider the following 3D elliptic problem:

$$-\frac{\partial}{\partial x_1}\left(k_1\frac{\partial u}{\partial x_1}\right) - \frac{\partial}{\partial x_2}\left(k_2\frac{\partial u}{\partial x_2}\right) - \frac{\partial}{\partial x_3}\left(k_3\frac{\partial u}{\partial x_3}\right) = f(x_1,x_2,x_3), \qquad \forall (x_1,x_2,x_3) \in \Omega, \tag{1}$$

$$0 < \sigma_{\min} \le k_1(x_1,x_2,x_3), \quad k_2(x_1,x_2,x_3), \quad k_3(x_1,x_2,x_3) \le \sigma_{\max},$$
$$u(x_1,x_2,x_3) = 0, \qquad \forall (x_1,x_2,x_3) \in \Gamma = \partial\Omega,$$

to be solved on the unit cube $[0,1]^3$. Let the domain be discretized by a uniform grid with n grid points in each coordinate direction.

A. Finite Difference Method

Let us consider the usual seven-point centered difference approximation for problem (1). This discretization leads to a system of linear algebraic equations

$$A\mathbf{x} = \mathbf{b}$$

where the vector of unknowns $\mathbf{x}$ has size n^3. If the grid points are ordered along the x_3 and x_2 directions first, the resulting matrix A admits a standard block-tridiagonal structure. Here, the diagonal blocks are block-tridiagonal matrices, while the

off-diagonal blocks are diagonal matrices. Overall, the matrix A can be written in the following form

$$A = tridiag(A_{i,i-1}, A_{i,i}, A_{i,i+1}) \qquad i = 1, 2, \ldots, n,$$

where $A_{i,i}$ are block-tridiagonal matrices which corresponds to one x_1-plane. For details see [3], [4], [5], [6].

III. Circulant Block-Factorization Preconditioning

Let us recall that a circulant matrix C has the form $(C_{k,j}) = \left(c_{(j-k) \bmod m}\right)$, where m is the size of C. Moreover, for any given coefficients $(c_0, c_1, \ldots, c_{m-1})$, let us denote by $C = (c_0, c_1, \ldots, c_{m-1})$ the circulant matrix

$$\begin{bmatrix} c_0 & c_1 & c_2 & \ldots & c_{m-1} \\ c_{m-1} & c_0 & c_1 & \ldots & c_{m-2} \\ \vdots & \vdots & \vdots & & \vdots \\ c_1 & c_2 & \ldots & c_{m-1} & c_0 \end{bmatrix}.$$

Any circulant matrix can be factorized as

$$C = F \Lambda F^*,$$

where Λ is a diagonal matrix containing the eigenvalues of C, F is the Fourier matrix

$$F = \frac{1}{\sqrt{m}} \left\{ e^{2\pi \frac{jk}{m} \mathbf{i}} \right\}_{0 \le j,k \le m-1}$$

and $F^* = \overline{F}^T$ denotes adjoint matrix of F. Here, $\mathbf{i}$ stands for the imaginary unit.

Let us now denote the general form of the CBF preconditioning matrix M, for the matrix A, by

$$M_{CBF} = tridiag(C_{i,i-1}, C_{i,i}, C_{i,i+1}) \qquad i = 1, 2, \ldots n$$

Here, $C_{i,j} = Block - Circulant(A_{i,j})$ is block-circulant approximation of the corresponding block $A_{i,j}$ [3], [4]. Note that the approach to defining block-circulant approximations can be interpreted as simultaneous averaging of the matrix coefficients, and changing the Dirichlet boundary conditions to the periodic ones.

Each PCG iteration consists of one solution of the linear system with the preconditioner. The CBF preconditioner can be written in the form

$$M_{CBF} = (I \otimes F \otimes F)(\Lambda \otimes I \otimes I)(I \otimes F^* \otimes F^*)$$

and the solution of the linear system with M_{CBF} requires one forward 2D Discrete Fourier Transform (DFT), solution of the tridiagonal linear systems, and one backward 2D DFT.

The details of the sequential and parallel realizations, of the CBF preconditioner, have been described in [5], [6], which should be consulted for the remaining details.

IV. Numerical Tests – Experimental Setup

Conducted experiments have been selected to illustrate the convergence rate, as well as the parallel performance of the developed algorithms for the 3D elliptic problems. Specifically, test problems, with variable coefficients in the form

$$\begin{aligned} \frac{\partial}{\partial x_1}\left[\left(1+\frac{\epsilon}{2}\sin\left(2\pi\left(x_1+x_3\right)\right)\right)\frac{\partial u}{\partial x_1}\right] \quad &+ \\ \frac{\partial}{\partial x_2}\left[\left(1+\frac{\epsilon}{2}\sin\left(2\pi\left(x_1+x_2\right)\right)\right)\frac{\partial u}{\partial x_2}\right] \quad &+ \\ \frac{\partial}{\partial x_3}\left[\left(1+\epsilon e^{x_1+x_2+x_3}\right)\frac{\partial u}{\partial x_3}\right] \quad &= \quad f\left(x_1,x_2,x_3\right) \end{aligned} \tag{2}$$

where $\epsilon \in [0, 1]$ is a parameter have been considered. It is well known that the circulant preconditioners are competitive with the incomplete LU factorization for moderately varying coefficients. This reflects the averaging of the coefficients, used in the block-circulant approximations.

The right hand side f, is chosen in such a way that the problem (2) has solution

$$u\left(x_1, x_2, x_3\right) = \sin 2\pi x_1 \sin 2\pi x_2 \sin 2\pi x_3.$$

All computations are done in double precision. The standard iteration stopping criterion is $||\mathbf{r}^{N_{it}}||_{M^{-1}}/||\mathbf{r}^0||_{M^{-1}} < 10^{-6}$, where $\mathbf{r}^j$ stands for the residual at the jth iteration step of the preconditioned conjugate gradient method. The code has been implemented in C. For the implementation of the preconditioning, Fast Fourier Transform (FFT) was used, and functions fftw_init_threads, fftw_plan_with_nthreads, fftw_plan_many_dft, and fftw_execute from the FFTW (the Fastest Fourier Transform in the West) library were used. A hybrid parallel code, based on joint application of MPI and OpenMP-based parallelizations has been developed [7], [8], [9], [10], [11].

In this contribution, the parallel code has been tested on cluster computer system Avitohol, at the Advanced Computing and Data Centre of the Institute of Information and Communication Technologies of the Bulgarian Academy of Sciences. The Avitohol consists of HP Cluster Platform SL250S GEN8. It has 150 servers, and two 8-core Intel Xeon E5-2650 v2 8C processors and two Intel Xeon Phi 7120P coprocessors per node. Each processor runs at 2.6 GHz. Processors within each node share 64 GB of memory. Each Intel Xeon Phi has 61 cores, runs at 1.238 GHz, and has 16 GB of memory. Nodes are interconnected with a high-speed InfiniBand FDR network (see, also http://www.hpc.acad.bg/).

For the experiments, Intel C compiler has been used and the code was compiled using the following options: "-O3 -qopenmp -L$(MKLROOT)/lib/intel64 -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -lpthread -lmkl_rt -lm" for the processors, and "-O3 -qopenmp -mmic -L$(MKLROOT)/lib/mic -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -lpthread -lmkl_rt -lm" for the coprocessors. Intel MPI was used to execute the code on the Avitohol computer system.

TABLE I
USED MEMORY, NUMBER OF ITERATIONS AND TIME (IN SECONDS) FOR THE EXECUTION OF THE PARALLEL ALGORITHM ON ONE NODE USING ONE MPI PROCESS AND VARYING THE NUMBER OF THREADS.

n	memory	N_{it}	Error	threads					
				1	2	4	8	16	32
120	707 Mb	57	7.7 E-3	8.53	7.68	4.80	4.13	3.81	7.42
240	3325 Mb	96	3.8 E-3	160.32	141.73	88.64	58.88	46.68	84.79
360	6631 Mb	132	2.6 E-3	771.61	732.17	423.83	276.96	218.15	372.30
480	15629 Mb	166	1.9 E-3	2425.11	2307.68	1302.72	832.37	671.93	1070.61
600	30458 Mb	200	1.5 E-3	6403.09	6336.52	3533.77	2155.12	1566.76	2598.90
720	52542 Mb	231	1.3 E-3	14192.50	13565.10	7753.98	4656.22	3346.86	5197.23

V. EXPERIMENTAL RESULTS AND ANALYSIS

The first series of experiments established the "baseline" performance. In Tables I and II results obtained using processors of a single node of Avitohol are presented. Table I shows the used memory, the number of iterations, the maximal error of the obtained solution, and the total execution time. The results have been obtained using only shared memory parallelism: i.e. there was one MPI process and up to 32 OpenMP threads.

The first observation concerns memory use. For n increasing from 120 to 720, memory consumption grows from 707 Mb to 52542 Mb. This means that the limit of size of the problem that can be solved on a single node has been reached. This was checked experimentally, and increasing n to 840 resulted in an "out of memory" error.

Second, let us note that for $n = 120$ there is no performance gain with the number of used threads. Clearly, problem is too small. However, for $n = 720$ speedup of order of 5 has been reached for 16 threads. This was also the "best result". Moving to 32 threads resulted in speedup decreasing to approximately 3. Interestingly, almost no performance improvement was observed when moving form 1 to 2 threads.

Approaching the performance from the"completely opposite" perspective, Table II shows the execution time when using only distributed memory parallelism; i.e. one OpenMP thread and up to 32 MPI processes. It should be noted that in the current (prototype) implementation, the algorithm works correctly only if the number of mesh points in each coordinate direction (n) is divisible by the number of processes. In the Table, the best execution time, for each size of the problem, is marked in bold.

It is easy to note that, for $n = 120, 240, \ldots 720$ the algorithm works much faster when using only MPI parallelism and utilizing multiple threads (in comparison to using OpenMP based multi-threading). For the largest case ($n = 720$), the solver that is using MPI and 30 threads is more than 2 times faster than using OpenMP and 16 threads (the fastest result from Table I). Overall, for the smallest problem ($n = 120$) the fastest execution is obtained when using 15 MPI processes. For bigger problems, on the other hand, the best results have been reached when using 30 processes.

TABLE II
EXECUTION TIME (IN SECONDS) FOR THE EXECUTION OF THE PARALLEL ALGORITHM ON ONE NODE USING ONE OPENMP THREAD.

n	processes			
	15	16	30	32
120	**1.68**		2.07	
240	26.02	24.50	**19.66**	
360	167.61		**83.46**	
480	532.95	589.49	**286.14**	306.16
600	1200.36		**709.44**	
720	2535.59	2385.89	**1834.67**	

The second series of experiments concerned use of individual processors. Specifically, Table III shows the execution time when using only processors from multiple nodes (from 2 to 8). Here, results for 1 OpenMP thread, as well as 16 and 32 threads are reported. Note that, results for $n = 960$ are also reported. This was possible due to the fact that the problem was "split" into at least two nodes and, therefore, it "fit in" (did not generate out of memory errors). Again, the best execution time, for each problem size, is marked in bold.

The algorithm runs the fastest when using 16 threads in just few cases, i.e. for $n = 120$ on 3 and 8 nodes, for $n = 240$ on 5 nodes, and for $n = 960$ on 2 nodes. In the remaining cases, the execution using one thread is the fastest. Considering the largest case ($n = 960$), for a single OpenMP thread, speedup of order 6 can be observed when moving from 2 to 8 nodes. Moreover, when comparing results with those reported in Table II, for $n = 720$, the best result on 8 nodes is more than 6 times faster.

In the next series of experiments, the performance of the co-processors has been evaluated [12]. Specifically, Tables IV and V present times collected on the Avitohol using only Intel Xeon Phi co-processors (processors have not been used for solving the computational problem). Here, Table IV shows the execution time on one co-processor for $n = 120, 240, 360$. Again, the best execution time is marked in bold.

Here, the positive effect of combining OpenMP and MPI based parallelism can be observed. For the largest problem that fit in the memory of the co-processor ($n = 320$), use of 4 OpenMP threads improved the performance by more than 4 times. This could be interpreted as a case of super-linear speedup. However, delving into this point is out of scope of this contribution.

Next, Table V presents execution times obtained when solving the problem on co-processors (only), but using up to 8 nodes. Note that, since the code is a prototype, case of 7 nodes had to be excluded. Here, again, it was possible, for

TABLE III
TIME (IN SECONDS) FOR THE EXECUTION OF THE PARALLEL ALGORITHM ON UP TO 8 NODES.

n	nodes											
	2		3		4		5		6		8	
	p_c	time	p_c	time	p_c	time	p_c	time	p_c	time	p_c	time
	1 OpenMP thread											
120	60	**1.21**	60	1.21	60	**0.52**	60	**0.58**	60	**0.47**	120	**0.43**
240	60	**10.81**	48	**10.49**	60	**8.92**	80	12.23	120	**5.80**	120	**4.68**
360	60	**44.53**	90	**43.79**	60	**37.17**	60	**33.91**	90	**35.54**	120	**32.14**
480	60	**138.40**	96	**115.16**	120	**104.98**	80	**102.40**	96	**94.00**	120	**80.36**
600	60	**336.15**	75	**264.43**	120	**213.35**	60	**229.31**	75	**193.97**	120	**149.82**
720	60	**766.71**	96	**865.68**	120	**389.33**	120	**453.28**	90	**405.31**	120	**292.54**
960	32	6552.93	96	**1985.21**	120	**1545.64**	160	**1597.70**	96	**1490.35**	96	**1047.86**
	16 OpenMP threads											
120	2	1.78	3	**1.08**	4	0.79	5	0.68	6	0.58	8	**0.43**
240	2	27.16	3	18.86	4	14.58	5	**11.85**	6	10.10	8	7.22
360	2	125.11	3	87.83	4	66.78	5	54.80	6	48.17	8	36.38
480	2	367.13	3	254.32	4	203.64	5	161.31	6	143.73	8	107.52
600	2	859.94	3	589.26	4	456.51	5	382.11	6	334.35	8	254.41
720	2	1643.03	3	1167.26	4	907.63	5	740.11	6	641.02	8	494.79
960	2	**6299.66**	3	3803.46	4	2957.54	5	7235.53	6	2081.50	8	1557.30
	32 OpenMP threads											
120	2	3.54	3	1.83	4	2.31	5	1.76	6	1.95	8	1.31
240	2	50.26	3	24.03	4	32.37	5	23.75	6	21.58	8	16.99
360	2	201.49	3	96.97	4	115.48	5	96.14	6	88.27	8	69.45
480	2	579.54	3	409.70	4	318.41	5	259.65	6	225.49	8	175.30
600	2	1371.68	3	948.50	4	722.42	5	585.15	6	501.13	8	385.27
720	2	2746.62	3	1893.08	4	1461.16	5	1181.41	6	1007.01	8	784.46
960	2	9611.23	3	5979.38	4	4526.58	5	9967.10	6	3053.44	8	2298.87

TABLE IV
EXECUTION TIME (IN SECONDS) FOR SOLVING OF 3D PROBLEM USING ONLY ONE CO-PROCESSOR OF THE AVITOHOL.

using one MPI process

n	threads				
	60	120	200	240	244
120	8.55	7.24	7.26	7.35	7.31
240	144.06	99.49	85.05	81.44	79.93
360	792.95	538.19	417.85	389.99	374.33

using p_m MPI processes and q_m OpenMP threads

n	p_m	q_m	time	p_m	q_m	time
120	120	1	2.86	120	2	**2.25**
240	120	1	104.63	60	4	**18.29**
360	120	1	170.97	60	4	**73.90**

larger number of nodes, to solve the problem for $n = 960$. In the Table, the best execution time is marked in bold.

It can be seen that the algorithm runs faster using 2 threads for $n = 120$ and 4 threads for $n = 240, 360$. In this context, it should be recalled that the memory of one co-processor is only 16 GB. This memory limit is the reason that the code could have been run only for small size problems. In particular, for problems with $n = 480$ at least 2 co-processors were needed, while for $n = 960$ the code could have been executed starting from 12 co-processors.

In the final series of experiments, processors and co-processors have been jointly used. Specifically, Table VI shows the best execution times collected on the Avitohol using Intel Xeon processors working together with the Intel Xeon Phi co-processors. Here, the code was executed using: on processors — p_c MPI processes and every process runs q_c OpenMP threads; on co-processors — p_m MPI processes and every process runs q_m OpenMP threads. In each case, the optimal combination of the number of MPI processes and the number of threads has been used. These combinations have been established experimentally. The memory limitation resulted in not being able to run experiments, for $n = 960$, for less than 3 nodes. For the reasons explained above, there are no results for 7 nodes.

As can be seen, due to the, above stated, memory limitations on co-processors, the largest problem size $n = 960$ required at least 3 nodes to be solved. Considering problem of size $n = 720$, use of 8 nodes turned out to be ineffective, as solution time increased, as compared to the use of 6 nodes. For 6 nodes an almost perfect speedup (larger than 5), has been obtained. Interestingly, for all problem sizes, use of 8 nodes resulted in performance that was inferior to 6 nodes. We do not have an explanation of this fact, other than possibility that in this case operations not related to the solution of the problem had to run "somewhere" and their execution interfered with execution of the solver. For the largest problem, when comparing the performance obtained on 3 and on 6 nodes, a speedup of almost 6 was recorded. This shows that if ample resources are provided, the proposed approach behaves as expected and parallelizes well, when applying hybrid approach to algorithm parallelization.

To better visualize the relationship between execution times, they have been visualized also in Figure 1. Here, the execution time of the hybrid code, on up to 8 nodes for

TABLE V
TIME (IN SECONDS) FOR THE EXECUTION OF THE PARALLEL ALGORITHM ON UP TO 8 NODES USING ONLY CO-PROCESSORS.

n	nodes											
	1			2			3			4		
	p_m	q_m	time	p_m	q_m	time	p_m	q_m	time	p_m	q_m	time
120	120	2	2.25	4	120	8.69	120	6	4.96	120	8	4.03
240	60	4	18.29	4	240	104.58	240	6	58.24	240	8	42.14
360	60	4	73.90	4	240	481.40	6	244	360.95	8	240	291.30
480	2	240	1615.20	4	240	1443.04	6	244	1077.12	8	200	870.12
600				4	200	3373.44	120	12	1809.61	120	16	1265.60
720				4	240	6746.71	120	6	3554.11	120	16	2673.83
	nodes											
	5			6			8					
	p_m	q_m	time	p_m	q_m	time	p_m	q_m	time			
120	120	10	3.44	120	24	3.78	15	120	3.96			
240	240	10	37.85	240	12	31.69	240	15	27.22			
360	10	200	253.00	12	240	225.09	15	200	187.94			
480	10	200	741.41	12	200	659.75	16	200	524.95			
600	120	10	1098.67	120	24	980.98	120	30	936.67			
720	10	200	3472.91	120	24	2018.62	120	30	1694.33			
960				120	24	6137.27	120	30	5234.65			

TABLE VI
EXECUTION TIME (IN SECONDS) FOR SOLVING OF 3D PROBLEM USING OPTIMAL COMBINATIONS OF PROCESSORS AND CO-PROCESSORS OF THE AVITOHOL.

n	nodes														
	1					2					3				
	p_c	q_c	p_m	q_m	time	p_c	q_c	p_m	q_m	time	p_c	q_c	p_m	q_m	time
120	2	8	2	120	7.15	32	1	28	34	3.62	48	1	12	60	2.05
240	2	8	2	244	79.71	32	1	28	34	54.16	6	8	6	120	43.82
360	30	1	30	17	305.37	32	1	14	17	210.11	6	8	6	244	198.51
480	30	1	30	17	828.02	32	1	28	34	620.84	48	1	48	30	466.77
600	30	1	30	17	1919.31	64	1	56	17	1378.84	48	1	27	24	943.17
720	2	8	2	244	5273.09	64	1	56	17	2630.10	48	1	42	34	2110.96
960						4	4	1	244	7702.80	48	1	48	30	6662.39
	4					6					8				
	p_c	q_c	p_m	q_m	time	p_c	q_c	p_m	q_m	time	p_c	q_c	p_m	q_m	time
120	8	8	7	240	2.57	96	1	24	60	1.55	64	2	56	30	3.03
240	128	1	112	8	30.49	192	1	28	30	23.40	128	1	112	17	16.36
360	8	8	7	244	171.41	12	8	12	244	115.17	15	8	15	120	95.26
480	8	8	8	244	477.08	12	8	12	244	341.25	16	8	16	240	268.53
600	64	1	56	34	810.39	96	1	54	48	525.54	15	8	15	240	648.94
720	64	1	56	34	1643.04	96	1	84	34	1193.33	16	16	16	244	1859.61
960	64	1	56	34	5230.96	96	1	96	30	3630.39	16	16	16	244	3725.99

$n = 240, 480, 960$, for CPU-only, co-processor only and when both CPU and co-processor were used. For each of these cases, results are represented using the same color and marking.

It can be seen that use of multiple nodes allows one to solve large problems. Nevertheless, the speedup, resulting from adding nodes is not overwhelming.

VI. CONCLUDING REMARKS

The aim of this contribution was to experimentally explore relationship between (1) 3D elliptic solver, based on preconditioned conjugate gradient, (2) its hybrid parallelization consisting of applying shared memory OpenMP threads and distributed memory MPI approach, and (3) complex supercomputer architecture, based on nodes, processors and co-processors. It has been established that memory availability is one of the key issues that strongly influences parallel performance. In this context it is difficult to apply standard performance measures, such as speedup, since largest problems require large number of nodes to be executed. However, even if a code can be executed on different number of nodes, adding more nodes may not result in performance gains. There is a "sweet spot" where the problem is executed the fastest and adding more resources does not help. This also means that potential for standard speedup is somewhat limited.

All these observations can be linked to complex interplay between hybrid parallelization and hybrid computer architecture. This may be also a warning sign that potential gains from hybrid approaches may be outweighed by losses caused by complexity of interactions between various "components".

ACKNOWLEDGMENTS

We acknowledge the provided access to the e-infrastructure of the National Centre for High Performance and Distributed Computing. This work has been accomplished with the partial support by the Grant No BG05M2OP001-1.001-0003, financed by the Science and Education for Smart Growth Operational Program (2014-2020) and co-financed by the European Union through the European structural and Investment

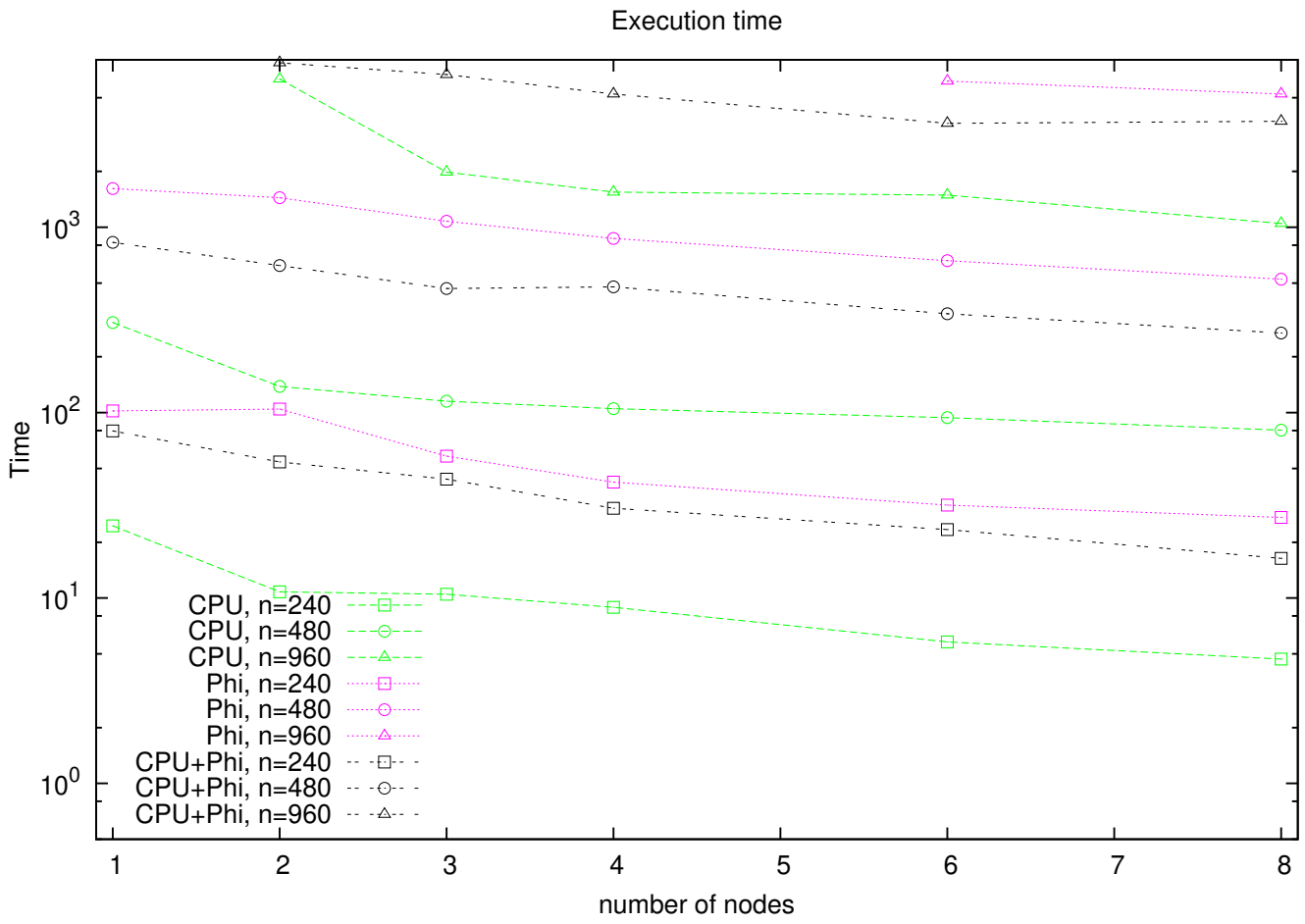

Fig. 1. Execution time for $n = 240, 480, 960$.

funds. Work presented here is a part of the collaborative grant between Polish Academy of Sciences and Bulgarian Academy of Sciences, IC-PL/01/2022-2023, "Practical aspects of scientific computing".

REFERENCES

[1] A. Axelsson and M. Neytcheva, *Supercomputers and numerical linear algebra*. Nijmegen: KUN, 1997.

[2] B. Bylina, J. Bylina, P. Stpiczyński, and D. Szałkowski, "Performance analysis of multicore and multinodal implementation of SpMV operation," in *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. Ganzha, L. Maciaszek, and M. Paprzycki, Eds., vol. 2. IEEE, 2014, pp. 569–576.

[3] I. Lirkov and Y. Vutov, "Parallel performance of a 3D elliptic solver," in *Proceedings of the International Multiconference on Computer Science and Information Technology*, M. Ganzha, M. Paprzycki, J. Wachowicz, and K. Węcel, Eds., vol. 1, 2006, pp. 579–590.

[4] ——, "The convergence rate and parallel performance of a 3D elliptic solver," *System Science*, vol. 32, no. 4, pp. 73–81, 2007.

[5] I. Lirkov and S. Margenov, "Parallel complexity of conjugate gradient method with circulant block-factorization preconditioners for 3D elliptic problems," in *Recent Advances in Numerical Methods and Applications*, O. Iliev, M. Kaschiev, B. Sendov, and P. Vassilevski, Eds. Singapore: World Scientific, 1999, pp. 482–490.

[6] I. Lirkov, S. Margenov, and M. Paprzycki, "Parallel performance of a 3d elliptic solver," in *Numerical Analysis and Its Applications II*, ser. Lecture Notes in Computer Science, L. Vulkov, J. Waśniewski, and P. Yalamov, Eds., vol. 1988. Springer, 2001, pp. 535–543.

[7] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan Kaufmann, 2000.

[8] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, ser. Scientific and engineering computation series. MIT press, 2008, vol. 10.

[9] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014.

[10] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, ser. Scientific and engineering computation series. Cambridge, Massachusetts: The MIT Press, 1997, second printing.

[11] D. Walker and J. Dongarra, "MPI: a standard Message Passing Interface," *Supercomputer*, vol. 63, pp. 56–68, 1996.

[12] F. Krużel and K. Banaś, "Finite element numerical integration on Xeon Phi coprocessor," in *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. Ganzha, L. Maciaszek, and M. Paprzycki, Eds., vol. 2. IEEE, 2014, pp. 603–612.