Performance Analysis of Hybrid Parallel Solver for 3D Stokes Equation on Intel Xeon Computer System

M. Ganzha,[1, a)] I. Lirkov,[2, b)] and M. Paprzycki[3, c)]

[1)] *Department of Mathematics and Information Science, Warsaw University of Technology, ul. Koszykowa 75, 00-662 Warsaw, Poland*

[2)] *Institute of Information and Communication Technologies, Bulgarian Academy of Sciences, Acad. G. Bonchev, bl. 25A, 1113 Sofia, Bulgaria*

[3)] *Systems Research Institute, Polish Academy of Sciences, ul. Newelska 6, 01-447 Warsaw, Poland*

[a)] *Electronic mail: M.Ganzha@mini.pw.edu.pl; http://pages.mini.pw.edu.pl/~ganzham/*

[b)] *Corresponding author:ivan@parallel.bas.bg; http://parallel.bas.bg/~ivan/*

[c)] *Electronic mail: marcin.paprzycki@ibspan.waw.pl; http://www.ibspan.waw.pl/~paprzyck/*

Abstract. In our previous work we have studied the performance of a parallel program, based on a direction splitting approach, solving time dependent Stokes equation. In it, we have used a rectangular uniform mesh, combined with a central difference scheme for the second derivatives. In our work, we were targeting massively parallel computers, as well as clusters of multi-core nodes. Therefore, the developed implementation used hybrid parallelization based on the MPI and OpenMP standards. Specifically, (i) between-node parallelism was supported by using MPI-based communication, while (ii) inside-node parallelism was supported by the OpenMP. In this way, by matching "structure of parallelization" with the architecture of modern large-scale computers, we have attempted at maximizing parallel efficiency of the program.
This paper presents an experimental performance study of the developed parallel implementation on a supercomputer using Intel Xeon processors, as well as Intel Xeon Phi co-processors. The experimental results show an essential improvement when running experiments for a variety of problem sizes and number of cores / threads.

INTRODUCTION

The objective of this paper is to analyze the parallel performance of a fractional time stepping technique, based on a direction splitting strategy, developed to solve the incompressible Navier-Stokes equations. Specifically, we consider the alternating directions algorithm, which was proposed in [1, 2]. Its key idea consists of using a projection scheme for solving unsteady Navier-Stokes equations. In this method, the pressure equation is derived from a perturbed form of the continuity equation, in which the incompressibility constraint is penalized in a negative norm induced by the direction splitting. The standard Poisson problem for the pressure correction is replaced by the series of one-dimensional second-order boundary value problems. This technique is proved to be stable and convergent (for all necessary details, see [1, 2]).

Initial results concerning parallel performance of the direction splitting algorithm for solving of 3D Stokes equation have been reported in [3]. Analysis of experimental results indicated that the algorithm is very well suited for distributed memory computers but its performance on a single multi-core node of a cluster was unsatisfactory. In other words, proposed implementation was not capable of taking advantage of multiple threads that could be running on individual cores within a single node. Next, in [4], we used LAPACK subroutines (see, [5]), from a multi-threaded library, for the solution of tridiagonal linear systems (which is the key computational component of the approach in question). The experimental results showed that the code needs additional improvements. Further, in [6] we have developed a hybrid parallel code based on combination of the MPI and the OpenMP standards [7, 8, 9, 10, 11]. In our parallel implementation of the partition method, each MPI process owned a small number of rows of the tridiagonal matrix, but the linear system has multiple right hand sides. Thus, in hybrid code, each OpenMP thread solves a tridiagonal system with a small number of rows and a small number of right hand side vectors. The performance of the hybrid parallel code (combining the MPI with the OpenMP) was highly efficient on a number of parallel computer systems with multi-core nodes.

The aim of this paper is to analyze parallel performance of the developed hybrid parallel code on a hybrid computer system, where each node consists of a multi-core Intel Xeon processors and many-core Intel Xeon Phi co-processors.

The remainder of the paper is organized as follows. The alternating directions algorithm, for the time-dependent Stokes equation is described in next section. Results of numerical tests are presented and analyzed in third section. Finally, some conclusions and future steps in our work are included in the last section.

Application of Mathematics in Technical and Natural Sciences
AIP Conf. Proc. 2164, 120003-1–120003-8; https://doi.org/10.1063/1.5130863
Published by AIP Publishing. 978-0-7354-1909-4/$30.00

ALTERNATING DIRECTIONS ALGORITHM FOR STOKES EQUATION

In our work, we consider the time-dependent Stokes equations written in terms of velocity $\mathbf{u}$ and pressure p:

$$\begin{cases} \mathbf{u}_t - \nu\Delta\mathbf{u} + \nabla p = \mathbf{f} & \text{in } \Omega\times(0,T) \\ \nabla\cdot\mathbf{u} = 0 & \text{in } \Omega\times(0,T) \\ \mathbf{u}|_{\partial\Omega} = 0, \quad \partial_n p|_{\partial\Omega} = 0 & \text{in } (0,T) \\ \mathbf{u}|_{t=0} = \mathbf{u}_0, \quad p|_{t=0} = p_0 & \text{in } \Omega \end{cases}, \tag{1}$$

where $\mathbf{f}$ is a smooth source term, ν is the kinematic viscosity, and $\mathbf{u}_0$ is a solenoidal initial velocity field, with a zero normal trace. The time interval $[0,T]$ is discretized on a uniform mesh and τ is the time step. We solve the problem (1) in the domain $\Omega = (0,1)^3$, for $t \in [0,T]$ with Dirichlet boundary conditions.

In 2010, Guermond and Minev [1, 2] introduced a novel fractional time stepping technique for solving the incompressible Navier-Stokes equations. This technique is based on a direction splitting strategy. They used a singular perturbation of the Stokes equation. In this way, the standard Poisson problem in the projection schemes was replaced by series of one-dimensional second-order boundary value problems.

Usage of central differences for the discretization in space, for the one-dimensional boundary value problems, leads to the solution of tridiagonal linear systems. Wang [12] proposed a partitioning algorithm for solving large tridiagonal systems of linear equations. The partitioning algorithm was primarily aimed at more coarse-grained parallel computation, where the number of processors is many times smaller than the number of unknowns. We have developed hybrid parallel code [6] based on an application of Wang's partitioning method for solving the tridiagonal system of linear equations, which arise in the direction splitting algorithm.

NUMERICAL EXPERIMENTS

To establish real-world characteristics of the proposed approach, we have considered time-dependent Stokes equations (1). We solved the problem in time interval $[0,2]$ and the discretization in time was done with time step $\tau = 10^{-2}$. The kinematic viscosity was $\nu = 10^{-3}$. The discretization in space used mesh sizes $h_1 = \frac{1}{n_1-1}$, $h_2 = \frac{1}{n_2-1}$, and $h_3 = \frac{1}{n_3-1}$ in x_1, x_2, and x_3 direction respectively.

Experimental setup

Let us now report on the experiments performed with the parallel implementation of the algorithm. A portable parallel code was implemented in C. As outlined above, the hybrid parallelization is based on joint application of the MPI and the OpenMP standards [7, 8, 9, 10, 11]. We use the LAPACK subroutines DPTTRF and DPTTS2 (see [5]) for solving tridiagonal systems of equations for the unknowns corresponding to the internal nodes in each sub-domain. The same subroutines are used to solve the reduced tridiagonal systems.

In our experiments, times were collected using the MPI provided timer. In all cases we report best times from multiple runs. However, it should be stressed that in all of our experiments resulted in times with very small variance in timings. In what follows, we report the elapsed time T_p (in seconds), when using m MPI processes and k OpenMP threads per MPI process. During the numerical experiments, we have tested our code on a single node, for the number of OpenMP threads varying from one to the maximal number of threads available on that node. On multiple nodes we report the best results for the number of OpenMP threads varying from the number of cores per node to the maximal available number of threads.

Let us denote the global number of threads by p. Keeping in mind all their well-known limitations, we report the parallel speed-up $S_p = T_1/T_p$ and the parallel efficiency $E_p = S_p/p$, as the simplest and easiest to grasp measures of parallel performance.

Experimental results

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.

TABLE I. Execution time (in seconds) for solving of 3D problem using only processors of a single node of the Avitohol.

n_1	n_2	n_3	k					
			1	2	4	8	16	32
176	176	176	428.34	364.45	207.58	117.09	76.85	47.50
176	176	352	919.95	781.84	437.26	243.64	165.37	133.79
176	352	352	2053.28	1978.74	952.83	511.12	342.74	204.86
352	352	352	4325.95	4095.75	2302.03	1116.27	731.64	464.67
352	352	704	9006.75	8222.57	4542.77	2412.82	1436.73	902.04
352	704	704	20658.20	19452.40	10476.10	5548.34	3091.60	1845.12
704	704	704	49080.20	46796.60	24264.30	12075.30	6506.90	3993.44

The computer system Avitohol is constructed with 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 co-processors 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. The maximal number of threads for one Intel Xeon processor is 16 (two threads per core), for one Intel Xeon Phi is 244 (four threads per core). Nodes are interconnected with a high-speed InfiniBand FDR network (for more details, see `http://www.hpc.acad.bg/`). We used the Intel C compiler, and compiled the code using the following options: "-O3 -qopenmp" for the processors and "-O3 -qopenmp -mmic" for the co-processors. To use the LAPACK subroutines, we linked our code to the optimized Intel Math Kernel Library (MKL). Intel MPI was used to execute the code on the Avitohol computer system.

Tables I and II present times collected on the Avitohol when using only the Intel Xeon processors to run the code.

Table I shows that using only processors on one node the best execution time is obtained using 32 OpenMP threads. We gain from the effect of hyper-threading for solving of discrete problems for all mesh sizes used in this set of experiments. Moreover, in all cases, adding more threads results in time reduction. Even for the smallest problem, increasing number of threads from 16 to 32 results in relative speed-up of 1.6. For the largest problem the same increase in the number of threads results in relative speed-up of 1.62. This indicates that, from the point of view of the solver, potential for gaining performance by increasing number of threads has not been exhausted. Finally let us observe that, for the smallest problem, the overall speed-up resulting from increasing number of threads from 1 to 32 is 9, while for the largest problem it increases to 12.3.

The execution time on 2 to 8 nodes (again, using only Xeon processors) is presented in Table II. Recall that results on a single node can be found in Table I. We report results for $k = 16$ and $k = 32$ threads. Here, slightly different results are observed. We used bold numbers to mark the best execution time, for 16 and 32 OpenMP threads for the corresponding number of nodes. The results show that, for up to 4 nodes, we gain from the effect of hyper-threading, using 32 threads, for all problems (mesh sizes). Increasing the number of nodes from 5 to 8 for the four "smallest"

TABLE II. Execution time (in seconds) for solving of 3D problem using only processors on many nodes of the Avitohol.

			nodes					
			2	3	4	5	6	8
n_1	n_2	n_3	k=16					
176	176	176	35.97	24.72	18.42	**15.81**	**10.89**	**8.22**
176	176	352	73.31	51.85	35.88	**35.09**	**23.96**	**19.05**
176	352	352	163.31	107.40	74.50	80.81	**52.67**	**37.32**
352	352	352	354.51	226.06	175.37	136.80	107.29	**81.17**
352	352	704	734.36	454.10	350.37	281.39	224.98	177.53
352	704	704	1541.51	884.62	738.06	558.83	455.97	367.33
704	704	704	3161.31	1928.48	1554.34	1410.72	924.94	782.11
n_1	n_2	n_3	k=32					
176	176	176	**33.73**	**23.25**	**14.16**	16.39	15.26	12.40
176	176	352	**52.08**	**38.59**	**34.76**	36.01	27.63	22.86
176	352	352	**103.65**	**97.65**	**50.82**	**77.42**	54.84	42.42
352	352	352	**242.25**	**181.51**	**141.41**	**125.73**	**106.19**	81.94
352	352	704	**471.27**	**358.89**	**273.91**	**258.27**	**196.69**	**147.93**
352	704	704	**951.30**	**607.79**	**494.84**	**428.49**	**392.48**	**301.39**
704	704	704	**1960.77**	**1192.85**	**949.38**	**908.34**	**611.76**	**526.62**

TABLE III. Speed-up for solving of 3D problem using only processors.

n_1	n_2	n_3	p										
			2	4	8	16	32	64	96	128	160	192	256
176	176	176	1.18	2.06	3.66	5.57	9.02	12.7	18.4	30.3	27.1	39.3	52.1
176	176	352	1.18	2.10	3.78	5.56	6.88	17.7	23.8	26.5	26.2	38.4	48.3
176	352	352	1.04	2.15	4.02	5.99	10.02	19.8	21.0	40.4	26.5	39.0	55.0
352	352	352	1.06	1.88	3.88	5.91	9.31	17.9	23.8	30.6	34.4	40.7	53.3
352	352	704	1.10	1.98	3.73	6.27	9.98	19.1	25.1	32.9	34.9	45.8	60.9
352	704	704	1.06	1.97	3.72	6.68	11.20	21.7	34.0	41.8	48.2	52.6	68.5
704	704	704	1.05	2.02	4.06	7.54	12.29	25.0	41.2	51.7	54.0	80.2	93.2

discrete problems (corresponding to coarse mesh size) the best time is obtained using 16 OpenMP threads. However, as the problem size increases, use of 32 threads becomes more competitive. For the three "largest" problems use of all available threads leads to best results. For the smallest problem, on 2 nodes there is almost no performance gain when increasing the number of threads from 16 to 32. However, for the largest problem, on 2 nodes, moving from 16 to 32 threads results in relative speed-up of 1.6. On 8 nodes, for the smallest problem, increasing number of threads from 16 to 32 results in a slowdown. However, for the largest problem, the relative speed-up is 1.48. In summary, it is clear that for the considered problems, and the computer we have run our experiments on, we have established limits of gains that can be achieved due to the effect of hyper-threading.

To provide an insight into performance of the parallel algorithm using only processors of the Avitohol, the obtained speed-up is reported in Table III. Let us note that in order to solve the problem with mesh sizes $n_1 = n_2 = n_3 = 704$ we need 60 GB of memory. Since the memory on one node is 64 GB it is the largest discrete problem that we can solve on one node. For larger problems we could not run the code on a single node and thus the speed-up could not be calculated.

It can be seen that on small number of cores largest speed-up is obtained for the smaller problems. Only starting from eight cores speed-up obtained on the largest problem starts to dominate. On eight nodes, for the smallest problem, efficiency of 20% is obtained. Efficiency increases with the problem size and reaches 55% for the largest problem reported.

Let us now consider what effects on performance has use of Xeon Phi co-processors. Tables IV and V present times collected on the Avitohol using only Intel Xeon Phi co-processors (no processors used for solving the computational problem).

It can be seen, for all problems, adding threads results in decrease of total execution time. This indicates, again, that for the proposed solution approach, the potential for performance gain when using OpenMP-based parallelization has not been exhausted. For the smallest problem, the speed-up resulting from using 244 threads is 54. For the largest problem, speed-up almost reaches 60. This means that in the largest case the efficiency is 25%.

Table V contains results obtained when running the code on up to eight nodes, but only using the Xeon Phi co-processor. Results reported in the table are for $k = 244$ threads. Here, let us note that the memory of one co-processor is 16 GB and the largest discrete problem that we can solve on one co-processor is for $n_1 = n_2 = n_3 = 352$. For larger problems we used at least two co-processors and for the problem with $n_1 = n_2 = n_3 = 704$ we need at least six co-processors (on three nodes).

Interestingly, our experiments (again) touched the limit of parallelization. Specifically, when executing the code on eight nodes, for the smallest discrete problem ($n_1 = n_2 = n_3 = 176$) the best execution time (23.07 seconds) was obtained using only 120 OpenMP threads (this result is not explicitly visible in Table V. It was only for the larger problems when use of $k = 244$ threads resulted in shortest execution time.

One can see the "strange" results when the parallel algorithm was executed on five nodes. Here, the performance

TABLE IV. Execution time (in seconds) for solving of 3D problem using only one co-processor of the Avitohol.

n_1	n_2	n_3	k						
			1	8	30	60	120	240	244
176	176	176	3966.12	1303.91	419.46	215.64	124.69	82.80	73.50
176	176	352	9633.71	2801.63	1005.47	461.99	258.81	179.14	154.44
176	352	352	20847.60	6908.14	2372.71	921.83	521.99	380.74	344.65
352	352	352	44186.10	18227.90	4387.60	2106.86	1182.09	771.75	737.53

TABLE V. Execution time (in seconds) for solving of 3D problem using only co-processors of the Avitohol ($k = 244$).

n_1	n_2	n_3	nodes						
			1	2	3	4	5	6	8
176	176	176	52.44	34.96	34.17	24.52	38.30	23.48	23.67
176	176	352	105.23	64.69	64.38	38.81	71.67	36.56	37.05
176	352	352	209.43	119.87	118.08	65.24	133.62	65.99	65.01
352	352	352	372.31	191.23	156.18	102.56	191.95	84.32	84.94
352	352	704	706.28	370.73	299.66	202.76	321.81	164.48	165.24
352	704	704		759.41	600.55	419.52	626.77	312.25	318.35
704	704	704			916.12	847.58	905.80	483.65	495.89

was worse than on four nodes. The reason for this is the parallelization approach used in our implementation (see, [3] for more details). In general, we decompose the computational domain into sub-domains and the number of sub-domains is equal to the number of MPI processes. Further, the entries of the vectors, corresponding of the mesh points in one sub-domain, are assigned to one MPI process. Using this approach, execution of the algorithm on four nodes leads to decomposition of the domain into $2 \times 2 \times 2$ sub-domains. Thus, solution of the one dimensional problems in the projection scheme require solving of linear systems with approximately the same size. In such way very good performance is achieved. On the contrary, running the algorithm on five nodes leads to decomposition of the domain into $5 \times 2 \times 1$ sub-domains. Small linear system are being solved in the x_1 direction, while large systems are solved in the x_3 direction. This is the reason for worse performance of the algorithm on five nodes.

Table VI shows the obtained speed-up of the parallel algorithm using only co-processors of the Avitohol. For the smallest case, speed-up on 8 nodes is 168, while for the largest case, where it was possible to solve the problem on a single node, speed-up was 520. For the largest solved problem, speed-up when increasing number of nodes from 4 to 8 was 1.8. This indicates that there is still some potential for further time reduction (by using more nodes).

It is clear that performance on eight nodes does not bring substantial performance gains over the performance on six nodes. As a matter of fact, for the smallest problem, the computing resources are "over-provisioned". Here, again, we can see that we are touching the limits of parallelization for the investigated problem and solution method.

Let us now consider what happens when full-blown hybrid parallelization is applied. Here, Table VII shows the best execution times collected on the Avitohol using Intel Xeon processors working together with the Intel Xeon Phi co-processors. In each case, the optimal number of threads has been used (see, Table VII for the details).

First, let us note that, again, performance on 5 nodes is worse than on 4. This happens for the same reasons as described above. Nevertheless, even for the smallest problem, wall-clock time reduction has been observed for up to 8 nodes.

For the smallest problem, the relative speed-up on eight nodes is 2.6. For the largest problem, on the other hand, the relative speed-up reaches 6.6. Therefore, for this series of experiments, limits of parallelization have not been reached (in particular, for the largest problem).

Table VIII presents the number of threads used to obtain the execution time in Table VII.

Here, we can see that for almost all problems and number of nodes the best results are obtained using 244 threads on Intel Xeon Phi. In general, for large problems the best performance is observed using 32 OpenMP threads on processors. For middle size problems, increasing the number of nodes and at the same time decreasing the number

TABLE VI. Speed-up for solving of 3D problem using only co-processors.

n_1	n_2	n_3	p						
			8	30	60	120	240	244	
176	176	176	3.04	9.46	18.39	31.81	47.90	53.96	
176	176	352	3.44	9.58	20.85	37.22	53.78	62.38	
176	352	352	3.02	8.79	22.62	39.94	54.76	60.49	
352	352	352	2.42	10.07	20.97	37.38	57.25	59.91	
n_1	n_2	n_3	p						
			488	976	1464	1952	2440	2928	3904
176	176	176	75.63	113.45	116.07	161.73	103.55	168.93	167.56
176	176	352	91.55	148.92	149.63	248.20	134.41	263.47	260.03
176	352	352	99.54	173.92	176.56	319.53	156.03	315.91	320.68
352	352	352	118.68	231.06	282.92	430.85	230.19	524.00	520.19

TABLE VII. Execution time (in seconds) for solving of 3D problem using processors and co-processors of the Avitohol.

n_1	n_2	n_3	nodes						
			1	2	3	4	5	6	8
176	176	176	37.45	24.60	20.97	20.96	22.06	17.80	14.23
176	176	352	67.24	42.16	34.01	33.55	37.06	27.34	21.98
176	352	352	127.26	72.73	60.15	60.15	66.57	44.56	35.20
352	352	352	244.83	131.32	87.28	86.72	89.56	61.92	53.73
352	352	704	468.64	253.22	162.65	160.53	190.32	120.29	100.89
352	704	704	956.65	493.84	311.13	308.42	311.95	229.65	171.68
704	704	704	1875.35	995.79	585.30	527.91	445.27	401.58	283.39

TABLE VIII. The number of threads used to obtain the best execution time in Table VII. We use the following notation: $m_c \times k_c + m_\varphi \times k_\varphi$ means m_c MPI processes on processors, k_c OpenMP threads on processors, m_φ MPI processes on coprocessors, k_φ OpenMP threads on coprocessors.

n_1	n_2	n_3	nodes						
			1	2	3	4	5	6	8
176	176	176	$2\times16+$	$4\times8+$	$6\times8+$	$8\times8+$	$10\times8+$	$12\times8+$	$16\times8+$
			2×244	4×240	6×240	8×244	10×120	12×244	16×120
176	176	352	$2\times16+$	$4\times16+$	$6\times8+$	$4\times16+$	$10\times8+$	$6\times16+$	$16\times8+$
			2×244	4×244	6×244	8×244	10×244	12×244	16×244
176	352	352	$2\times16+$	$4\times16+$	$6\times8+$	$8\times8+$	$10\times8+$	$6\times16+$	$16\times8+$
			2×244	4×244	6×244	8×244	10×244	12×244	16×244
352	352	352	$2\times16+$	$4\times16+$	$6\times16+$	$4\times16+$	$10\times8+$	$6\times16+$	$16\times8+$
			2×244	4×244	6×244	8×244	10×244	12×244	16×244
352	352	704	$2\times16+$	$4\times16+$	$6\times16+$	$8\times16+$	$10\times16+$	$6\times16+$	$16\times16+$
			2×244	4×240	6×244	8×244	10×244	12×244	16×244
352	704	704	$2\times16+$	$4\times16+$	$6\times16+$	$8\times16+$	$10\times16+$	$6\times32+$	$16\times16+$
			2×244	4×244	6×240	8×244	10×244	12×244	16×244
704	704	704	$4\times8+$	$4\times16+$	$3\times32+$	$8\times16+$	$10\times16+$	$12\times16+$	$16\times16+$
			2×244	4×244	6×244	8×244	10×244	12×244	16×244

of OpenMP threads to 16 ensures better performance. Finally, for small problems the best performance is observed using 16 OpenMP threads on processors.

Let us also compare the performance when solving the problem only on processors, only on co-processors and using hybrid approach. To be able to see directly compare results spread in the tables above, we have created Table IX in which we have brought results (from tables above) for up to eight nodes for (i) only processors (for the best number of threads), (ii) only co-processors (for the best number of threads), and (iii) the hybrid approach (for the best number of threads).

TABLE IX. Execution time for $n_1 = n_2 = n_3 = 176, 352, 704$

n_1	n_2	n_3	nodes						
			1	2	3	4	5	6	8
only processors									
176	176	176	47.50	33.73	23.25	14.16	15.81	10.89	8.22
352	352	352	464.67	242.25	181.51	141.41	125.73	106.19	81.17
704	704	704	1960.77	3993.44	1192.85	949.38	908.34	611.76	526.62
only co-processors									
176	176	176	52.44	34.96	34.17	24.52	38.30	23.48	23.07
352	352	352	372.31	191.23	156.18	102.56	191.95	84.32	84.94
704	704	704			916.12	847.58	905.80	483.65	495.89
hybrid approach									
176	176	176	37.45	24.60	20.97	20.96	22.06	17.80	14.23
352	352	352	244.83	131.32	87.28	86.72	89.56	61.92	53.73
704	704	704	1875.35	995.79	585.30	527.91	445.27	401.58	283.39

From the table we can see that for smallest problem, for number of nodes from four to eight, use of only processors outperforms the remaining two approaches. This seems to be the clear case of Amdahl's effect. The smallest problem is too small for the number of threads (and nodes) applied to it.

Situation changes when the two larger problems are considered. Here, the hybrid approach is a clear winner. This seems to indicate that if the computing system consists of Xeon processors and Phi co-processors, then there may be reasons to use only processors, or to use hybrid architecture. However, there does not seem to be a reason (at least for the type of problems we are interested in) to use only co-processors.

Finally, we compare the performance of the parallel algorithm using only processors, only co-processors, and using both processors and co-processors. The best time obtained on up to eight nodes for three discrete problems is shown in Figure 1.

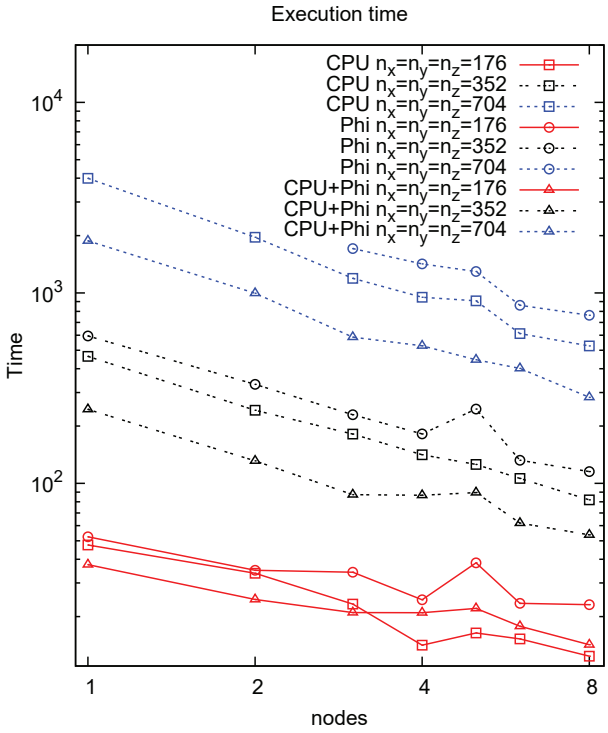

FIGURE 1. Execution time for $n_1 = n_2 = n_3 = 176, 352, 704$

CONCLUDING REMARKS AND FUTURE WORK

We have studied the parallel performance of a hybrid implementation of the direction splitting algorithm for solving of the 3D time dependent Stokes equation on a finite time interval and on a uniform rectangular mesh. This paper is an extension of our previous work and consist of an experimental performance study of a parallel implementation on a supercomputer using Intel Xeon processors as well as Intel Xeon Phi co-processors. The experimental results show that when problems are small, using Xeon processors together with Phi co-processors may result in over-provisioning of computing power. Moreover, it is rather difficult to justify use of only co-processors. Finally, when problems are large enough, use of hybrid architecture results in clear advantage.

In the current version of our parallel implementation, the computational domain is decomposed into sub-domains. The sizes of all sub-domains are almost the same. In order to tune the implementation and to have good performance we tested the algorithm running various number of MPI processes on processors while using one MPI process per co-processor. As a next step, in order to achieve better load balance on the hybrid architecture, we have to make

further changes in the MPI code. In this way we can avoid or at least minimize the delay caused by different load in the Intel Xeon processors and Intel Xeon Phi co-processors.

ACKNOWLEDGMENTS

We acknowledge the provided access to the e-infrastructure of the National centre for high performance and distributed computing. This research was partially supported by grant KP-06-N27/6 from the Bulgarian NSF. 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 funds. This research was part of the collaboration agreement between Bulgarian Academy of Sciences and Polish Academy of Sciences "Practical aspects of scientific computing".

REFERENCES

1. J.-L. Guermond and P. Minev (2010) *Comptes Rendus Mathematique* **348**, 581–585.
2. J.-L. Guermond and P. Minev (2011) *Computer Methods in Applied Mechanics and Engineering* **200**, 2083–2093.
3. M. Ganzha, K. Georgiev, I. Lirkov, S. Margenov, and M. Paprzycki, "Highly parallel alternating directions algorithm for time dependent problems," in *AMiTaNS'11*, AIP CP**1404**, edited by M.D. Todorov and C.I. Christov (American Institute of Physics, Melville, NY, 2011), pp. 210–217.
4. I. Lirkov, M. Paprzycki, M. Ganzha, and P. Gepner, "Performance evaluation of MPI/OpenMP algorithm for 3D time dependent problems," in *Preprints of Position Papers of the Federated Conference on Computer Science and Information Systems, Annals of Computer Science and Information Systems*, Vol. **1**, edited by M. Ganzha, L. Maciaszek, and M. Paprzycki (2013), pp. 27–32.
5. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd edn (SIAM, Philadelphia, 1999).
6. M. Ganzha, K. Georgiev, I. Lirkov, and M. Paprzycki (2015) *Computers & Mathematics with Applications* **70**, 2762–2772.
7. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference, Scientific and Engineering Computation Series* (The MIT Press, Cambridge, Massachusetts, 1997) second printing.
8. D. Walker and J. Dongarra (1996) *Supercomputer* **63**, 56–68.
9. W. Gropp, E. Lusk, and A. Skjellum *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (The MIT Press, 2014).
10. R. Chandra, R.Menon, L. Dagum, D. Kohr, D.Maydan, and J.McDonald, *Parallel Programming in OpenMP* (Morgan Kaufmann, 2000).
11. B. Chapman, G. Jost, and R. Van Der Pas, Using OpenMP: *Portable Shared Memory Parallel Programming*, Vol. 10 (MIT press, 2008).
12. H. Wang (1981) *ACM Transactions on Mathematical Software (TOMS)* **7**, 170–183.