An application of partition method for solving 3D Stokes equation

Maria Ganzha[a], Krassimir Georgiev[b], Ivan Lirkov[b,*], Marcin Paprzycki[a]

[a]*Systems Research Institute, Polish Academy of Sciences*
ul. Newelska 6, 01-447 Warsaw, Poland
[b]*Institute of Information and Communication Technologies*
Bulgarian Academy of Sciences
Acad. G. Bonchev, bl. 25A
1113 Sofia, Bulgaria

Abstract

In our previous work we have studied the performance of a parallel algorithm, based on a direction splitting approach, for solving of time dependent Stokes equation. We used a rectangular uniform mesh, combined with a central difference scheme for the second derivatives. Hence, the proposed algorithm required only solution of tridiagonal linear systems.

In our work, we are targeting massively parallel computers, as well as clusters of multi-core nodes. The somehow slower (experimentally-established) performance of the proposed approach was observed when using all cores on a single node of a cluster. To remedy this problem, we tried to use LAPACK subroutines from the multi-threaded layer library, but the parallel performance of the code (while improved) was still not satisfactory on a single (multi-core) node.

Our current work considers hybrid parallelization based on the MPI and OpenMP standards. It is motivated by the need to maximize the parallel efficiency of our implementation of the proposed algorithm. Essential improvements of the parallel algorithm are achieved by introducing two levels of parallelism: (i) between-node parallelism based on the MPI and (ii) inside-node parallelism based on the OpenMP. The implementation was tested on Linux clusters with Intel processors and on the IBM supercomputer.

Keywords: Navier-Stokes, time splitting, ADI, incompressible flows, parallel algorithm
2010 MSC: 35Q30, 35Q35, 65F05, 65F50, 65N06, 68W10,

*Corresponding author
Email addresses: `maria.ganzha@ibspan.waw.pl` (Maria Ganzha), `georgiev@parallel.bas.bg` (Krassimir Georgiev), `ivan@parallel.bas.bg` (Ivan Lirkov), `paprzyck@ibspan.waw.pl` (Marcin Paprzycki)
URL: `http://inf.ug.edu.pl/~mganzha/` (Maria Ganzha), `http://parallel.bas.bg/~georgiev` (Krassimir Georgiev), `http://parallel.bas.bg/~ivan/` (Ivan Lirkov), `http://www.ibspan.waw.pl/~paprzyck/` (Marcin Paprzycki)

1. Introduction

The solution of a tridiagonal system of linear equations lies at the heart of many programs developed for, so called, scientific computations. With the development and availability of multitude of parallel and vector computers, parallel algorithms (suitable for these machines) have appeared also for solving tridiagonal systems of equations.

Large tridiagonal systems of linear equations appear in many numerical applications. For instance, in [1], they arise in line relaxations needed by robust multigrid methods for structured grid problems. In [2] adaptive mesh refinement algorithm was used for a coupled system of nonlinear evolution equations of a hyperbolic type and a parallel algorithm was used to solve the tridiagonal systems of linear equations. The above papers used the classic parallel algorithm called the "partition method" introduced in [3].

On a serial computer, Gaussian elimination without pivoting can be used to solve a diagonally dominant tridiagonal system of linear equations. This algorithm, first described in [4], is commonly referred to as the Thomas algorithm. Unfortunately, this algorithm is not well suited for parallel computers. The first parallel algorithm for the solution of tridiagonal systems was described in [5]. It is now usually referred to as cyclic reduction. Stone introduced his recursive doubling algorithm in [6]. Both cyclic reduction and recursive doubling are designed for fine grained parallelism, where each processor owns exactly one row of the tridiagonal matrix. Wang proposed a partitioning algorithm that was aimed at more coarse-grained parallel computation, where the number of processors is many times smaller than the number of unknowns [3]. Diagonal dominance of the resulting reduced system in Wang's method was established in [7] and numerical stability of Wang's algorithm was analyzed in [8]. A unified approach for the derivation and analysis of partitioning methods applicable to solution of tridiagonal linear systems was given in [9, 10]. There has also been attention directed towards a parallel partitioning of the standard LU algorithm. Sun et al. [11] introduced the parallel partitioning LU algorithm that is very similar to the Bondeli's divide and conquer algorithm [12]. For both the partitioning algorithms and the divide and conquer algorithms, a reduced tridiagonal system of interface equations must be solved. Here, each processor owns only a small number of rows in this reduced system. As an example, in Wang's partitioning algorithm, each processor owns one row of the reduced system. In [13], this reduced system is solved by recursive doubling. However, numerical experiments were performed only on a very small numbers of processors (typical for the times that this contribution was published).

Austin et al. [1] targeted parallel computers with thousands (to tens of thousands) of processors, such that for a 2D structured grid, line solves spanning hundreds of processors are realistic. They represent a memory efficient partitioning algorithm, for the solution of diagonally dominant tridiagonal linear systems of equations. This partitioning algorithm is well suited for current distributed memory parallel computers.

2. Alternating directions algorithm for Stokes equation

We consider the time-dependent Stokes equations written in terms of velocity $\mathbf{u}$ and pressure p:

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

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 $t \in [0,2]$ with Dirichlet boundary conditions.

Guermond and Minev [14, 15] 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. In our original research we developed MPI code based on an application of the partition method for solving the tridiagonal system of linear equations, which arise in the alternating directions algorithm [16, 17]. The analysis of experimental results showed that the algorithm is very well suited for distributed memory parallel computers but it has unsatisfactory performance on a single (multi-core) node of a cluster. To try to alleviate this deficiency, we used LAPACK subroutines from a multi-threaded layer library, for the solution of tridiagonal linear systems [18]. The experimental results showed that the code needs additional improvements. Here, one has to recall that to maximize performance of a cluster of multi-core nodes, one has to, first, optimize the per-node performance.

In the current work, we have developed a hybrid-parallel code based on combination of the MPI and the OpenMP standards. In our application of the partition method, each MPI process owns a small number of rows of the tridiagonal matrix, but the linear system has multiple right hand sides. In our hybrid implementation, each OpenMP thread solves the tridiagonal system with a small number of rows and a small number of right hand side (RHS) vectors. Specifically, let us consider a discretization, in space, with n_x, n_y, and n_z points in direction x, y, and z respectively. Then the one-dimensional problem in the x direction leads to a linear system with n_x rows and $3n_y n_z$ RHS vectors for the "velocity update" step and $n_y n_z$ vectors for the "penalty" step (in the alternating directions algorithm [15]). We use $m = m_x m_y m_z$ MPI processes and k OpenMP threads. In the "penalty" step, each MPI process computes the coefficients in $\frac{n_x}{m_x}$ rows and $\frac{n_y}{m_y}\frac{n_z}{m_z}$ RHS vectors. Let us denote by M the

HPCG	Galera	MareNostrum	IBM Blue Gene/P
Compiler			
Intel C Compiler 12.1.0			IBM XL C Compiler 9.0
MPI			
Intel MPI 4.0.3.008	OpenMPI 1.4.3	Intel MPI 4.1.3.049	MPICH2
LAPACK			
Intel Math Kernel Library 11.1			Engineering and Scientific Subroutine Library 5.1

Table 1: Compilers and libraries used on the four computer systems

number of rows and by K the number of RHS vectors owned by a single MPI process. In our current implementation each OpenMP thread solves a linear system with M rows and $\frac{K}{k}$ RHS vectors.

3. Experimental results

Let us now report on the experiments we have performed with the current implementation of the solver. In the experiments, we consider the time-dependent Stokes equations (1). The discretization in time was done with time step 10^{-2}. The kinematic viscosity was $\nu = 10^{-3}$. The discretization in space used mesh sizes $h_x = \frac{1}{n_x-1}$, $h_y = \frac{1}{n_y-1}$, and $h_z = \frac{1}{n_z-1}$. The total number of unknowns in the discrete problem was $800\, n_x\, n_y\, n_z$.

To solve the problem, a portable parallel code was designed and implemented in C. As stated above, the hybrid parallelization is based on joint application of the MPI and the OpenMP standards [19, 20, 21, 22]. In the code, we use the LAPACK subroutines DPTTRF and DPTTS2 (see [23]) 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.

The parallel code has been tested on a cluster computer system HPCG located in the Institute of Information and Communication Technologies, on a cluster computer system Galera, located in the Polish computing center TASK, on a supercomputer MareNostrum at the Barcelona Supercomputing Center, and on the IBM Blue Gene/P machine at the Bulgarian Supercomputing Center. Table 1 summarizes the information about compilers and libraries used on the four computer systems. In our experiments, times have been collected using the MPI provided timer and we report the best results from multiple runs. In what follows, we report the elapsed time T_p (in seconds), when using m MPI processes and k OpenMP threads, where $p = m \times k$, the parallel speed-up $S_p = T_1/T_p$, and the parallel efficiency $E_p = S_p/p$.

Tables 2 and 3 show the results collected on the HPCG cluster. The HP Cluster Platform Express 7000 consists of 36 blades BL 280c, with dual Intel Xeon X5560 processors (for the total of 288 cores and 576 computational threads).

n_x	n_y	n_z	k				
			1	2	4	8	16
100	100	100	87.86	40.19	23.89	18.11	18.28
100	100	200	199.73	90.04	49.63	37.37	35.82
100	200	200	412.11	182.33	109.77	77.86	74.61
200	200	200	908.53	403.13	229.51	169.14	145.67
200	200	400	1898.00	832.34	460.76	320.36	305.80
200	400	400	3171.97	1716.60	953.55	659.17	631.28
400	400	400	6570.93	3599.35	2000.45	1368.45	1329.70
400	400	800	14529.50	7910.66	4383.55	2889.60	2907.80

Table 2: Execution times of solving the 3D problem on a single node of the HPCG.

Each processor runs at 2.8 GHz. Processors within each blade share 24 GB RAM, while nodes are interconnected with non-blocking DDR Interconnection via the Voltaire Grid director 2004 with latency 2.5 μs and bandwidth 20 Gbps (see also `http://www.grid.bas.bg/hpcg/`). We used an Intel C compiler, and compiled the code with the option "`-O3 -openmp`". For solving the tridiagonal systems of equations using LAPACK subroutines, we linked our code to the Intel Math Kernel Library (MKL). Initial results collected on the HPCG, originating from the MPI code linked to the multi-threaded Intel MKL, for solving the linear systems, were published in [18]. Here, Tables 2 and 3 show the results from the hybrid code using OpenMP for solving the linear systems (within a multi-core node). Let us recall that one HPCG node has two processors with eight physical cores that can run a code with 16 OpenMP threads using hyper-threading. This is why we report also performance values for $k = 16$. The execution time presented in Table 2 shows that it is possible to obtain some performance gain when using hyper-threading (using 2 OpenMP threads per physical core). However, this effect is not across-the-board. For instance, it is easy to see that we gain very little from the effect of hyper-threading in the case of linear systems with 200–400 rows (see, also Table 3). It is clear that deficiencies in memory management hamper performance when $k = 16$ threads are used to solve the largest problems.

Fig. 1 shows a comparison between the last two versions of the parallel code: (i) the previous version, using multi-threaded Intel MKL for solving of linear systems (for more details, see [18]) and, (ii) the new one using OpenMP for solving the linear systems. It is relatively clear that the current approach outperforms the previous one "across the board" of tried problem sizes, numbers of nodes and cores. To provide an analytic view of the performance, the speed-up obtained on the HPCG is reported in Table 4. Here, note that the largest discrete problem that we could solve on a single blade has 128×10^6 grid points. Furthermore, problems larger than these reported in the table did not fit into the available memory (of a given number of nodes). For the largest problems reported in Table 4, efficiency is slightly above 50%, e.g. for the largest case overall; $400 \times 400 \times 800$, for $p = 128$ it reaches about 53%.

			nodes			
			2	4	8	16
n_x	n_y	n_z	$k = 8$			
100	100	100	9.72	5.02	3.01	1.92
100	100	200	18.53	10.42	5.49	4.09
100	200	200	40.00	19.18	14.42	7.46
200	200	200	82.57	46.28	27.02	14.78
200	200	400	165.91	89.73	50.20	30.77
200	400	400	338.66	177.60	110.31	63.17
400	400	400	679.60	367.18	180.12	106.58
400	400	800	1327.64	722.13	379.21	216.08
400	800	800	2951.39	1486.41	773.56	423.47
800	800	800		3345.61	1623.03	855.05
800	800	1600			3412.26	1658.27
n_x	n_y	n_z	$k = 16$			
100	100	100	8.70	4.85	3.88	2.87
100	100	200	19.79	10.44	6.85	5.32
100	200	200	38.88	19.65	12.13	9.07
200	200	200	79.32	41.54	22.56	12.70
200	200	400	164.07	78.43	41.78	25.27
200	400	400	318.29	158.46	80.30	46.67
400	400	400	668.37	321.32	167.18	88.88
400	400	800	1306.57	641.91	339.30	184.43
400	800	800	3108.37	1334.47	689.03	361.78
800	800	800		3380.27	1560.22	926.30
800	800	1600			3371.75	1827.10

Table 3: Execution times of solving the 3D problem on many nodes of the HPCG.

n_x	n_y	n_z	$m \times k$						
			2	4	8	16	32	64	128
100	100	100	2.19	3.68	4.85	9.03	17.49	29.21	45.64
100	100	200	2.22	4.02	5.35	10.78	19.17	36.40	48.79
100	200	200	2.26	3.75	5.29	10.30	21.48	28.58	55.25
200	200	200	2.25	3.96	5.37	11.00	19.63	33.62	61.45
200	200	400	2.28	4.12	5.92	11.44	21.15	37.81	61.68
200	400	400	1.85	3.33	4.81	9.37	17.86	28.76	50.22
400	400	400	1.83	3.28	4.80	9.67	17.90	36.48	61.65
400	400	800	1.84	3.31	5.03	10.94	20.12	38.32	67.24

Table 4: Speed-up on the HPCG.

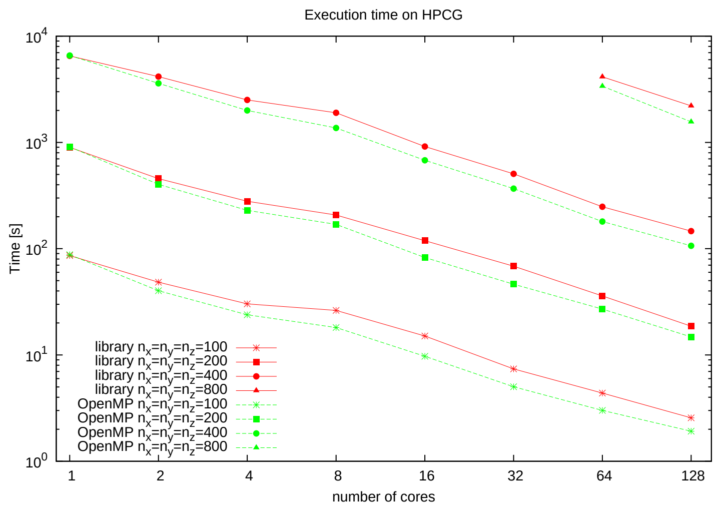

Figure 1: Execution times of the two versions of the code on the HPCG.

n_x	n_y	n_z	k			
			1	2	4	8
100	100	100	172.74	98.79	67.53	57.54
100	100	200	370.96	211.34	143.28	129.28
100	200	200	831.13	482.54	325.89	281.96
200	200	200	1731.85	994.65	680.83	575.11
200	200	400	3476.49	2040.48	1397.81	1175.16
200	400	400	7137.60	4200.39	2855.00	2374.29
400	400	400	14609.90	8424.69	5865.67	4845.97
400	400	800	29444.40	16885.00	11975.70	10093.00

Table 5: Execution times of solving the 3D problem on a single node of the Galera.

Tables 5 and 6 show the results collected on the Galera cluster. It is a Linux cluster with 336 nodes, and two Intel Xeon quad core processors per node. Each processor runs at 2.33 GHz. Processors within each node share 8, 16, or 32 GB of memory. Nodes are interconnected with a high-speed InfiniBand network (see also `http://www.task.gda.pl/kdm/sprzet/Galera`). When running our code on the Galera, we used the Intel C compiler, and compiled the code with the options "`-O3 -openmp`". To use the LAPACK subroutines, we linked our code to the Intel MKL. The results on the Galera, collected when running the MPI code linked to the optimized multi-threaded Intel MKL, for solving the tridiagonal linear systems, were also published in [18]. Here, Tables 5 and 6 show the results from the hybrid code using the OpenMP for solving the linear systems. Again, the discrete problem with $n_x = n_y = 400, \quad n_z = 800$ requires 22 GB of memory. That is why, for larger problems, we could not run the code

n_x	n_y	n_z	nodes							
			2	4	8	16	32	64	128	256
100	100	100	29.2	13.7	7.0	4.3	2.6	1.9	2.2	1.7
100	100	200	63.3	29.0	14.1	8.0	4.6	2.6	2.1	2.1
100	200	200	139.1	64.0	31.1	16.5	8.4	4.6	3.7	3.0
200	200	200	288.6	132.2	62.1	32.9	15.7	8.0	5.5	4.0
200	200	400	586.3	278.4	138.2	72.4	33.8	15.9	10.4	6.4
200	400	400	1197.1	590.8	300.9	161.4	74.4	34.5	22.4	11.9
400	400	400	2395.2	1194.4	626.0	318.6	148.8	67.8	39.5	19.3
400	400	800	4780.9	2448.7	1273.2	651.5	316.9	150.9	90.5	45.1
400	800	800	10562	4815.9	2843.7	1336.9	664.1	329.0	197.3	95.1
800	800	800		11165	5008.4	2789.6	1389.0	668.0	359.1	181.3
800	800	1600			14764	8955.9	2970.2	1427.4	755.8	382.6
800	1600	1600				15637	5008.5	2985.8	1609.7	762.1
1600	1600	1600					15318	8302.4	3452.7	1584.9
1600	1600	3200						20063	14609	3678.3

Table 6: Execution times of solving the 3D problem on many nodes of the Galera.

for the small number of nodes (it did not fit into the available memory). As a matter of fact, the largest problem, reported in Table 6, requires a minimum of 64 nodes to be solved.

As can be seen, on a single node of Galera, using OpenMP allows to take advantage of 2 threads per processor (note that $k = 4$ means that two processors within a node were running two threads each). Furthermore, while the actual gain, when moving from $k = 4$ to $k = 8$ for the largest case, provides only about 16% performance improvement; in the real wall-clock time this is more than 30 minutes of actual run-time reduction. This latter number shows that the actual performance gain (considered from the perspective of the user who is "awaiting results") is substantial, nevertheless.

Fig. 2 shows a comparison of the execution times, on the Galera, between the last two versions of the parallel code: using multi-threaded Intel MKL for solving the linear systems, and using OpenMP for solving them. Again, the improved code, considered in this paper, outperforms the previous one across the board (for the larger problems). Note that time is reported using a logarithmic scale.

Table 7 contains the speed-up obtained on Galera. For the reasons described above, it was impossible to calculate the standard speed-up (time on a single core vs. time on p cores). That is why we report the speed-up on Galera only for the problems with $n_x, n_y = 100, 200, 400, \quad n_z = 100, 200, 400, 800$. However it is easy to calculate the "normalized" speed-up even for the largest problem. For instance, when comparing execution time on 128 and 256 cores, with that on 64 cores, for $n_x = n_y = n_z = 1600$ one can say that such speed-up is 2.405 and 5.238 respectively. Here, the super-linear speed-up can be attributed to the well-known effect caused by halving the problem size and improving memory management. These results indicate also that the communication in the parallel

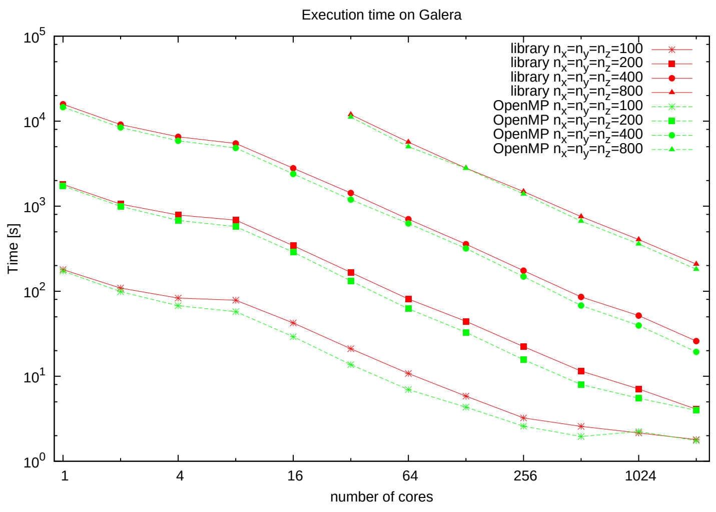

Figure 2: Execution times of the two versions of the code on the Galera.

n_x	n_y	n_z	$m \times k$										
			2	4	8	16	32	64	128	256	512	1024	2048
100	100	100	1.75	2.56	3.00	5.92	12.63	24.80	39.86	66.78	88.46	77.08	98.77
100	100	200	1.76	2.59	2.87	5.86	12.80	26.29	46.12	81.21	140.08	177.58	175.12
100	200	200	1.72	2.55	2.95	5.98	12.98	26.75	50.25	99.22	180.87	223.65	272.43
200	200	200	1.74	2.54	3.01	6.00	13.10	27.88	52.70	110.61	216.79	314.53	435.54
200	200	400	1.70	2.49	2.96	5.93	12.49	25.16	47.99	102.89	218.00	332.65	543.20
200	400	400	1.70	2.50	3.01	5.96	12.08	23.72	44.23	95.92	206.72	318.90	599.96
400	400	400	1.73	2.49	3.01	6.10	12.23	23.34	45.86	98.15	215.41	369.85	755.35
400	400	800	1.74	2.46	2.92	6.16	12.02	23.13	45.20	92.92	195.14	325.29	653.10

Table 7: Speed-up on the Galera.

algorithm is mainly local. Specifically, if halving the problem leads to super-linear speed-up, it means that communication between nodes (facilitated by calls to the MPI primitives) is not as important as memory contention within a node. For the largest case, for which we were able to fit the problem on a single node, the efficiency of the performance for $p = 2048$ is about 31%. Interestingly, when comparing the results for $p = 128$ reported in Table 7 with these reported in Table 4, one can see that the HPCG machine reaches substantially higher parallel efficiency (35% vs. 53%).

Tables 8, 9, 10, 11, 12, and 13 show the results collected on the MareNostrum, the most powerful supercomputer in Spain. It has 3,056 compute nodes, and two Intel SandyBridge 8-core processors per node. Each processor runs at 2.6 GHz. Processors within each node share 32, 64, or 128 GB of memory. Nodes are interconnected with a high-speed InfiniBand FDR10 network (see also `http://`

n_x	n_y	n_z	k				
			1	2	4	8	16
100	100	100	55.67	34.12	21.29	15.61	13.57
100	100	200	120.01	73.28	46.63	33.58	28.64
100	200	200	303.67	180.39	109.94	77.30	62.77
200	200	200	646.36	390.89	239.14	158.96	128.48
200	200	400	1318.25	801.08	489.24	323.87	250.90
200	400	400	2918.51	1813.99	1024.21	652.63	516.24
400	400	400	6133.14	3763.26	2120.11	1333.70	1002.37
400	400	800	15278.10	8139.33	4585.25	2814.30	2061.91
400	800	800	30456.10	17545.70	9583.50	5999.65	4319.39
800	800	800	82170.30				

Table 8: Execution times of solving the 3D problem on a single node of the MareNostrum using multi-threaded layer library for solving the linear systems.

n_x	n_y	n_z	nodes						
			2	4	8	16	32	64	128
100	100	100	7.05	3.84	2.20	1.55	1.16	0.91	2.88
100	100	200	14.17	7.18	4.01	2.86	1.68	1.26	2.88
100	200	200	29.62	14.60	7.65	4.91	2.88	1.86	5.41
200	200	200	64.58	30.22	15.13	8.42	4.71	2.96	7.18
200	200	400	128.76	60.34	31.02	16.62	8.77	5.30	6.60
200	400	400	259.00	130.47	67.00	32.56	16.93	9.51	10.88
400	400	400	517.45	258.18	134.22	65.00	33.69	17.61	15.02
400	400	800	1047.55	539.52	260.59	135.19	66.61	35.01	24.66
400	800	800	2151.33	1075.80	527.48	288.99	139.85	73.20	45.00
800	800	800	4367.25	2127.17	1050.29	570.65	274.10	143.43	81.41
800	800	1600	9024.27	4478.21	2161.97	1176.59	572.33	277.95	183.30
800	1600	1600		9158.31	4487.02	2354.38	1126.30	558.01	371.42
1600	1600	1600			9040.61	4671.52	2257.58	1098.03	705.93

Table 9: Execution times of solving the 3D problem on many nodes of the MareNostrum using multi-threaded Intel MKL.

n_x	n_y	n_z	$m \times k$										
			2	4	8	16	32	64	128	256	512	1024	2048
100	100	100	1.63	2.62	3.57	4.10	7.90	14.50	25.36	35.9	50.9	61.2	19.3
100	100	200	1.64	2.57	3.57	4.19	8.47	16.70	29.91	41.9	71.3	95.3	41.7
100	200	200	1.68	2.76	3.93	4.84	10.25	20.80	39.69	61.8	105.4	163.7	56.1
200	200	200	1.65	2.70	4.07	5.03	10.01	21.39	42.71	76.7	137.3	218.6	90.0
200	200	400	1.65	2.69	4.07	5.25	10.24	21.85	42.50	79.3	150.4	248.6	199.7
200	400	400	1.61	2.85	4.47	5.65	11.27	22.37	43.56	89.5	172.4	306.7	268.3
400	400	400	1.63	2.89	4.60	6.12	11.85	23.75	45.69	94.4	182.0	348.2	408.3
400	400	800	1.88	3.33	5.43	7.41	14.58	28.32	58.63	113.0	229.4	436.4	619.7
400	800	800	1.74	3.18	5.08	7.05	14.16	28.31	57.74	105.4	217.8	416.1	676.8
800	800	800					18.82	38.63	78.24	143.9	299.8	572.9	1009.4

Table 10: Speed-up on the MareNostrum using the multi-threaded Intel MKL.

n_x	n_y	n_z	k				
			1	2	4	8	16
100	100	100	56.72	31.73	16.26	9.71	7.38
100	100	200	120.78	66.09	36.12	22.04	16.11
100	200	200	303.93	169.93	90.51	54.29	37.59
200	200	200	657.86	359.78	202.08	114.64	79.97
200	200	400	1365.13	754.74	417.60	238.22	157.10
200	400	400	3050.03	1712.71	886.85	484.13	317.52
400	400	400	6424.91	3572.48	1841.49	991.14	635.04
400	400	800	17766.80	7761.12	4033.29	2157.43	1343.79
400	800	800	30452.10	16752.70	8484.86	4676.80	2874.20
800	800	800	81398.50				

Table 11: Execution times of solving the 3D problem on a single node of the MareNostrum using OpenMP for solving the linear systems.

n_x	n_y	n_z	nodes						
			2	4	8	16	32	64	128
100	100	100	3.99	2.26	1.55	1.23	0.96	0.79	3.69
100	100	200	8.02	4.20	2.55	2.06	1.32	1.02	5.07
100	200	200	17.45	8.37	4.60	3.33	2.00	1.47	6.46
200	200	200	39.97	17.91	8.66	5.23	3.04	2.11	5.29
200	200	400	81.37	35.98	18.22	10.43	5.52	3.50	5.10
200	400	400	167.05	80.98	42.17	20.18	10.49	6.16	6.02
400	400	400	326.71	163.37	84.07	40.14	20.79	11.16	10.03
400	400	800	690.09	347.05	164.31	87.49	41.04	22.15	19.07
400	800	800	1421.90	687.08	340.43	188.25	90.45	47.63	29.90
800	800	800	2934.58	1417.05	671.78	376.33	178.01	93.76	52.09
800	800	1600	6386.14	3036.08	1436.45	792.65	381.07	182.18	126.41
800	1600	1600		6554.86	2996.60	1574.35	755.64	364.60	242.45
1600	1600	1600			6801.71	3201.66	1504.29	721.48	462.74

Table 12: Execution times of solving the 3D problem on many nodes of the MareNostrum using OpenMP for solving the linear systems.

n_x	n_y	n_z	$m \times k$										
			2	4	8	16	32	64	128	256	512	1024	2048
100	100	100	1.79	3.49	5.84	7.69	14.22	25.12	39.42	46.1	64.9	71.4	15.4
100	100	200	1.83	3.34	5.48	7.50	15.06	28.77	48.04	58.8	101.2	118.8	23.8
100	200	200	1.79	3.36	5.60	8.08	17.42	36.33	66.04	91.2	151.9	206.8	47.1
200	200	200	1.83	3.26	5.74	8.23	16.46	36.74	75.92	125.8	216.9	311.5	124.5
200	200	400	1.81	3.27	5.73	8.69	16.78	37.94	74.94	131.0	247.2	389.9	267.5
200	400	400	1.78	3.44	6.30	9.61	18.26	37.67	72.32	151.1	290.8	494.9	506.6
400	400	400	1.80	3.49	6.48	10.12	19.67	39.33	76.43	160.1	309.1	575.6	640.3
400	400	800	2.29	4.41	8.24	13.22	25.75	51.19	108.13	203.1	432.9	802.2	931.8
400	800	800	1.82	3.59	6.51	10.59	21.42	44.32	90.01	161.8	336.7	639.3	1018.4
800	800	800				27.74	57.44	121.17	216.3	457.3	868.2	1562.7	

Table 13: Speed-up on the MareNostrum using OpenMP for solving the linear systems.

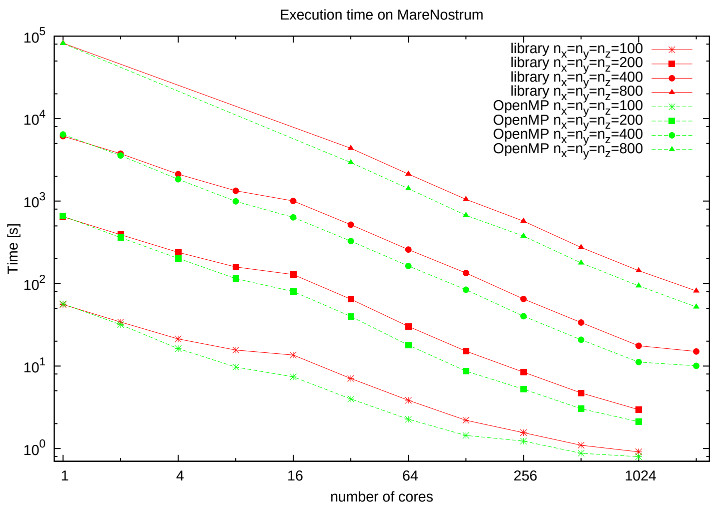

Figure 3: Execution time of the two versions of the code on the MareNostrum.

`www.bsc.es/marenostrum-support-services/mn3`). When running our code on the MareNostrum, we used the Intel C compiler, and compiled the code with the options "`-O3 -openmp`". To use the LAPACK subroutines, we linked our code to the optimized Intel MKL. The execution time in Tables 8 and 9 were obtained using multi-threaded Intel MKL for solving the linear systems. The results in Tables 11 and 12 were obtained when using OpenMP for solving the linear systems.

Comparing the results in Tables 9 and 12 one can see that the code using OpenMP is almost twice faster than the code which uses the MKL library, for solving the linear systems. This fact can be observed also in Fig. 3, which illustrates the execution times on the MareNostrum.

Parallel efficiency for $p = 512$, for the largest problem reported in Table 13 is quite high (about 89%). The parallel efficiency for $p = 128$ is also the highest among machines considered thus far (about 95%). Furthermore, the approach advocated here is faster across the board (Fig. 3). This is particularly visible for the largest problems depicted there.

Tables 14 and 15 present times collected on the IBM Blue Gene/P supercomputer. For our experiments we used the BG/P machine located at the Bulgarian Supercomputing Center. The supercomputer has two BG/P racks. One BG/P rack consists of 1024 compute nodes with quad core PowerPC 450 processors (running at 850 MHz). Each node has 2 GB of RAM. For the point-to-point communications a 3.4 Gb 3D mesh network is used (for more details, see `http://www.scc.acad.bg/`). In our experiments, we have used the IBM XL C compiler and compiled the code with the following options: "`-O5 -qstrict -qarch=450d -qtune=450 -qsmp=omp`". To use the LAPACK subroutines, we linked our code to the Engineering and Scientific Subroutine Library (ESSL).

n_x	n_y	n_z	k		
			1	2	4
100	100	100	881.59	468.72	239.50
100	100	200	1812.58	946.49	493.11
100	200	200	3779.01	1963.90	1014.33
200	200	200	7664.79	3963.43	2149.08

Table 14: Execution times of solving the 3D problem on a single node of the IBM Blue Gene/P.

k=4							
n_x	n_y	n_z	nodes				
			2	4	8	16	32
100	100	100	111.71	56.17	28.62	15.85	8.41
100	100	200	239.66	119.95	57.80	30.91	15.98
100	200	200	483.06	244.59	122.30	59.83	29.91
200	200	200	1032.40	498.51	250.14	127.00	60.34
200	200	400	2079.17	1035.53	501.06	261.43	127.81
200	400	400		2067.20	1047.61	510.72	256.84
400	400	400			2141.69	1094.20	531.46
400	400	800				2216.74	1096.65
400	800	800					2173.92
			nodes				
			64	128	256	512	1024
100	100	100	4.88	3.46	2.45	1.69	1.56
100	100	200	8.65	6.17	4.26	2.58	2.36
100	200	200	16.22	11.41	6.89	4.12	3.81
200	200	200	29.68	19.22	11.59	6.90	5.09
200	200	400	61.52	36.47	22.15	11.91	8.95
200	400	400	122.20	72.37	39.41	21.46	16.39
400	400	400	261.89	135.18	73.72	37.57	25.51
400	400	800	534.29	284.94	153.23	74.47	47.85
400	800	800	1088.32	577.08	297.02	143.91	93.37
800	800	800	2275.78	1154.25	588.49	295.28	161.66
800	800	1600		2315.33	1222.41	592.77	349.23
800	1600	1600			2374.26	1193.87	701.46
1600	1600	1600				2443.41	1299.52

Table 15: Execution times of solving the 3D problem on many nodes of the IBM Blue Gene/P.

n_x	n_y	n_z	$m \times k$					
			2	4	8	16	32	64
100	100	100	1.88	3.68	7.89	15.69	30.80	55.62
100	100	200	1.92	3.68	7.56	15.11	31.36	58.65
100	200	200	1.92	3.73	7.82	15.45	30.90	63.17
200	200	200	1.93	3.57	7.42	15.38	30.64	60.35
			$m \times k$					
			128	256	512	1024	2048	4096
100	100	100	104.78	180.69	254.47	359.97	520.3	566.4
100	100	200	113.40	209.60	293.94	425.16	703.7	767.5
100	200	200	126.34	232.99	331.19	548.47	916.4	991.3
200	200	200	127.03	258.27	398.80	661.27	1110.7	1507.2

Table 16: Speed-up on the IBM Blue Gene/P.

Here we have to note that one node of the BG/P machine that we have used, has only 2 GB of RAM. That is why on a single node we could run the code only for $n_x, n_y, n_z \leq 200$. Table 16 contains the speed-up on the BG/P for the problems, which could have been solved on a single node. On 128 nodes the parallel efficiency is 82–99% and on 256 nodes a super-linear speed-up is observed (with the same reasons for this effect as these discussed above). To estimate the efficiency of the code for larger problems one can calculate the, so called, weak scalability (defined as how the solution time varies with the number of processors for a fixed problem size per processor). For example, the weak scalability for problems with 10^6 grid points per compute node is between 81.1% and 99.9% on up to 512 compute nodes.

Fig. 4 shows a comparison of the execution times on the IBM Blue Gene/P obtained by the last two versions of the parallel code: using multi-threaded Intel MKL for solving the linear systems and using OpenMP for solving the linear systems. As on the remaining machines, the improved approach outperforms the previous-best-one "across the board." As a matter of fact, on the Blue Gene the performance gain is the most pronounced of all computers we have run our experiments on.

Finally, in Fig. 5, we represent execution time of the hybrid code on all four systems. Results are presented for problems of size $n_x = n_y = n_z = 100, 200, 400, 800$. Here, it can be seen that the MareNostrum is the most efficient computer architecture for our problem. Observe also that one core of the HPCG (and of the MareNostrum) is about 12–15 times faster than one core of the Blue Gene/P. The situation changes as the number of cores increases. For 128 cores the clusters are only about 6–7 times faster than the supercomputer. It is clear that using more than 64 nodes on the Galera cluster, for the smallest problem with $n_x = n_y = n_z = 100$, results in a case of Amdahl's effect (where adding more resources does not result in a time reduction). This is not the case for the BG/P machine, where the relative "weakness" of a "relatively old" processor is significantly compensated by the efficiency of its networking

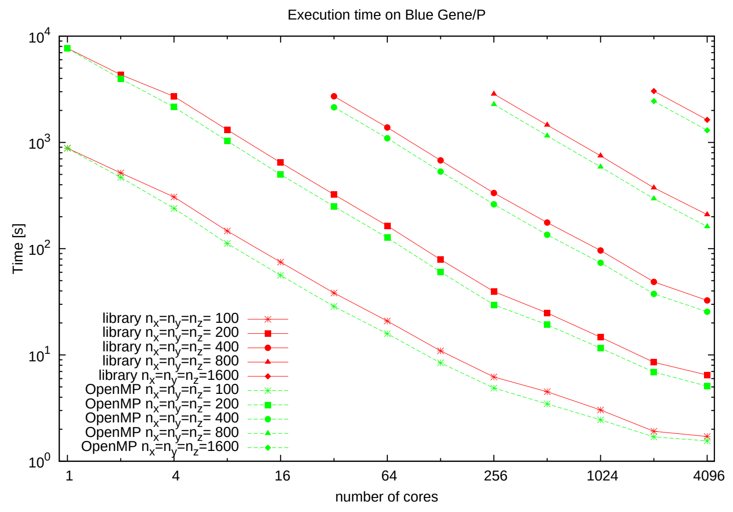

Figure 4: Execution times of the two versions of the code on the IBM Blue Gene/P.

infrastructure.

4. Concluding remarks

We have implemented an improved, hybrid-parallel version of the partition method for solving of a tridiagonal system of linear equations, which arise in the alternating directions algorithm used for solving a class of Navier-Stokes equations. Specifically, our implementation was based on combining the MPI and OpenMP standards. In our hybrid implementation, each MPI process owns a small number of rows of the tridiagonal matrix, while each OpenMP thread solves the tridiagonal system with a small number of rows and a small number of right hand side vectors. The experimental results show an essential improvement of the hybrid parallel code, combining the MPI with the OpenMP, over the previous implementation; when running experiments for a variety of problem sizes and number of cores / threads on four distinct parallel computer systems.

Acknowledgments

Computer time grants from the TASK computing center in Gdansk, Poland, the Bulgarian Supercomputing Center (BGSC), and the Barcelona Supercomputing Center are kindly acknowledged. This research was partially supported by grant I01/5 from the Bulgarian NSF. Work presented here is a part of the Poland-Bulgaria collaborative grant “Parallel and distributed computing practices”.

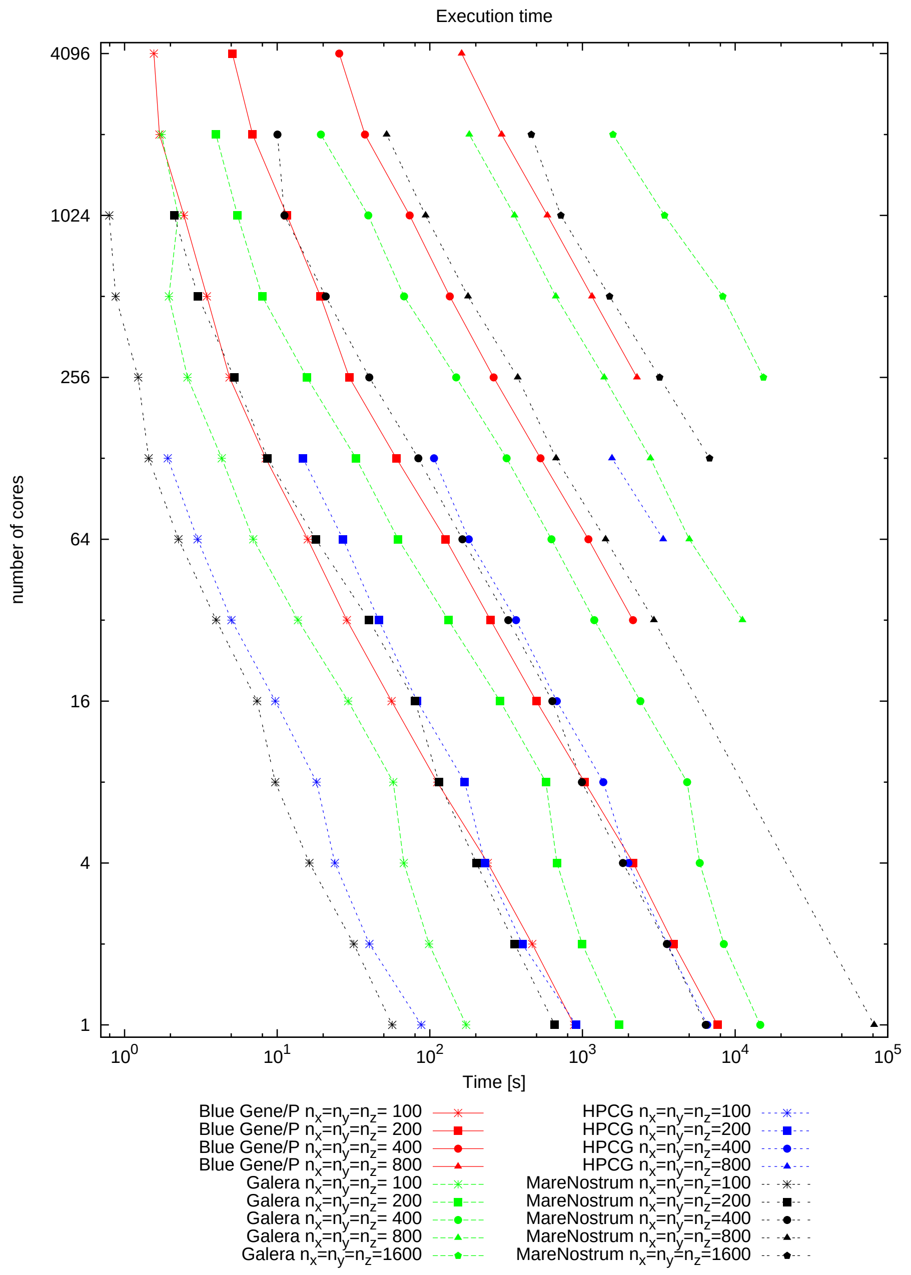

Figure 5: Execution time for $n_x = n_y = n_z = 100, 200, 400, 800$.

References

[1] T. M. Austin, M. Berndt, J. D. Moulton, A memory efficient parallel tridiagonal solver, Preprint LA-VR-03-4149 (2004).

[2] E. A. Hayryan, J. Busa, E. E. Donets, I. Pokorny, O. I. Strel'tsova, Numerical studies of perturbed static solutions decay in the coupled system of Yang-Mills-dilaton equations with use of MPI technology, Tech. rep., Laboratory of Information Technologies, Joint Institute for Nuclear Research, Dubna (Russian Federation) (2004).

[3] H. Wang, A parallel method for tridiagonal equations, ACM Transactions on Mathematical Software (TOMS) 7 (2) (1981) 170–183.

[4] L. H. Thomas, Elliptic problems in linear difference equations over a network, Watson Sci. Comput. Lab. Rept., Columbia University, New York.

[5] R. Hockney, O. Buneman, A fast direct solution of Poissons equation using fourier analysis, Communications of the ACM 6 (7) (1963) 357–357.

[6] H. S. Stone, An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, Journal of the ACM (JACM) 20 (1) (1973) 27–38.

[7] C. H. Walshaw, Diagonal dominance in the parallel partition method for tridiagonal systems, SIAM journal on matrix analysis and applications 16 (4) (1995) 1086–1099.

[8] P. Yalamov, V. Pavlov, On the stability of a partitioning algorithm for tridiagonal systems, SIAM journal on matrix analysis and applications 20 (1) (1998) 159–181.

[9] P. Amodio, L. Brugnano, Parallel factorizations and parallel solvers for tridiagonal linear systems, Linear algebra and its applications 172 (1992) 347–364.

[10] P. Amodio, L. Brugnano, T. Politi, Parallel factorizations for tridiagonal matrices, SIAM journal on numerical analysis 30 (3) (1993) 813–823.

[11] X.-H. Sun, H. Z. Sun, L. M. Ni, Parallel algorithms for solution of tridiagonal systems on multicomputers, in: Proceedings of the 3rd international conference on Supercomputing, ACM, 1989, pp. 303–312.

[12] S. Bondeli, Divide and conquer: A parallel algorithm for the solution of a tridiagonal linear system of equations, Parallel Computing 17 (4) (1991) 419–434.

[13] J. Hofhaus, E. F. Van de Velde, Alternating-direction line-relaxation methods on multicomputers, SIAM Journal on Scientific Computing 17 (2) (1996) 454–478.

[14] J.-L. Guermond, P. Minev, A new class of fractional step techniques for the incompressible Navier-Stokes equations using direction splitting, Comptes Rendus Mathematique 348 (9–10) (2010) 581–585.

[15] J.-L. Guermond, P. Minev, A new class of massively parallel direction splitting for the incompressible Navier-Stokes equations, Computer Methods in Applied Mechanics and Engineering 200 (23) (2011) 2083–2093.

[16] M. Ganzha, K. Georgiev, I. Lirkov, S. Margenov, M. Paprzycki, Highly parallel alternating directions algorithm for time dependent problems, in: C. Christov, M. Todorov (Eds.), Applications of Mathematics in Technical and Natural Sciences, AMiTaNS 2011, Vol. 1404 of AIP Conference Proceedings, 2011, pp. 210–217.

[17] I. Lirkov, M. Paprzycki, M. Ganzha, Performance analysis of parallel alternating directions algorithm for time dependent problems, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Waśniewski (Eds.), 9th international conference on Parallel Processing and Applied Mathematics, PPAM 2011, Part I, Vol. 7203 of Lecture notes in computer science, Springer, 2012, pp. 173–182.

[18] I. Lirkov, M. Paprzycki, M. Ganzha, P. Gepner, Performance evaluation of MPI/OpenMP algorithm for 3D time dependent problems, in: M. Ganzha, L. Maciaszek, M. Paprzycki (Eds.), Preprints of Position Papers of the Federated Conference on Computer Science and Information Systems, Vol. 2 of Annals of Computer Science and Information Systems, 2013, pp. 27–32.

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

[20] D. Walker, J. Dongarra, MPI: a standard Message Passing Interface, Supercomputer 63 (1996) 56–68.

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

[22] B. Chapman, G. Jost, R. Van Der Pas, Using OpenMP: portable shared memory parallel programming, Vol. 10, MIT press, 2008.

[23] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, 3rd Edition, SIAM, Philadelphia, 1999.