Parallel Computational Models to Estimate an Actual Speedup of Analyzed Algorithm

Igor Konshin[1,2,3(✉)]

[1] Dorodnicyn Computing Centre of the Russian Academy of Sciences, Moscow 119333, Russia
Igor.Konshin@gmail.com
[2] Institute of Numerical Mathematics of the Russian Academy of Sciences, Moscow 119333, Russia
[3] Lomonosov Moscow State University, Moscow 119991, Russia

Abstract. The paper presents two models of parallel program runs on platforms with shared and distributed memory. By means of these models, we can estimate the speedup when running on a particular computer system. To estimate the speedup of OpenMP program the first model applies the Amdahl's law. The second model uses properties of the analyzed algorithm, such as algorithm arithmetic and communication complexities. To estimate speedup the computer arithmetic performance and data transfer rate are used. For some algorithms, such as the preconditioned conjugate gradient method, the speedup estimations were obtained, as well as numerical experiments were performed to compare the actual and theoretically predicted speedups.

Keywords: Parallel computations · Computational complexity · Communication complexity · Speedup

1 Introduction

During the last decades a parallel computing has been the basic tool for the solution of the most time consuming problems of mathematical physics, linear algebra, and many other branches of modern supercomputer application [1]. The most fitted parallel computational model allows to adequately estimate the numerical algorithms efficiency. It gives a possibility to compare the performance of the analyzed algorithms for the concrete architectures and choose the most successful ones in advance.

There are a lot of parallel computation models (see, for example, [1–3]), however some of them are too superficial to estimate the quantitative speedup values, on the contrary the other requires to take into account too detailed information on the algorithm and a run of code implementation. Moreover, the most of the models do not reflect the peculiarity of the architectures of the computers, on which the implemented algorithms are running. Using only macrostructure of algorithms there would be interesting to decide which algorithm

V. Voevodin and S. Sobolev (Eds.): RuSCDays 2016, CCIS 687, pp. 304–317, 2016.
DOI: 10.1007/978-3-319-55669-7_24

properties are the most important for deriving of practical quantitative speedup estimates and which computation systems characteristics could be applied to get these estimates.

To analyze an algorithm properties for different computer architectures, for example, for computers with the shared or distributed memory fitted computational models may be required. While analyzing algorithms for each computational model a knowledge of the conditions and regions of the model applicability becomes very important.

This paper describes two parallel computation models, presents the efficiency estimations obtained on their base, and defines the conditions of their applications. Specification and analysis of the parallel efficiency upper bounds estimates are performed for some linear algebra algorithms, including the preconditioned conjugate gradient method. The qualitative comparison of estimates obtained and results of numerical experiments are presented.

2 Parallel Computational Model for the Shared Memory Computers

Let the parallel computations be performed on a shared memory computer and programming environment OpenMP be used for parallelization. In the most simple cases, OpenMP can be treated as an insertion of compiler directives for loops parallelization. At some intermediate parallelization stage or due to intrinsic algorithm properties some arithmetic operations can be performed in a serial mode.

Let us consider that the computations are sufficiently uniform by the set of arithmetic operations performed and, in principle, we can calculate the amount of such operations and obtain the total numbers of parallel and serial operations. Let the fraction of serial operation be $f \in [0; 1]$ (this value sometimes is denoted by "s" from "serial" but we are not doing so to separate the usage of "S" for speedup).

If the time T for the arithmetic operations is linear with respect to the number of arithmetic operations, then it is easy to estimate the maximum speedup, which can be obtained for some implementation of the considered algorithm.

2.1 Amdahl's Law

Let denote by $T(p)$ the time of program running on p processors, than the speedup received for computations using p processors will be expressed by classical formula:

$$S(p) = T(1)/T(p). \tag{1}$$

If the fraction of serial operations is equal to f, then using formula (1) we can estimate the maximum achievable speedup by the following way:

$$S(p) = T(1)/(fT(1) + (1 - f)T(1)/p)) = p/(1 + f(p - 1)). \tag{2}$$

The last formula expresses the Amdahl's law [5]. It can also be treated in more general case as the maximum achieved speedup which can be obtained for arbitrary parallel architecture for the analyzed algorithm or the program code [6].

For the analysis of the obtained formula (2) it can be noted that if the fraction of serial operations is just 1%, then when running the program on 100 processors, the serial part of the code on a single processor will take about the same time as the parallel part of operations on 100 processors. It results in about 50 speedup (or 50% of efficiency). If the number of processors used can be arbitrarily increased, then the maximum achieved speedup will be equal to $S = 1/f$. Another extreme case is the linear speedup $S(p) = p$ achieved for $f = 0$.

In addition, we can define the best conditions of the Amdahl's law applicability for a shared memory computer, that is, when the algorithm actual speedup will be close to the estimated by (2). The basic conditions are:

- the arithmetic operations are quite uniform;
- all the used threads take part in the computations of the parallel part of the code;
- load balancing for all active threads;
- scalability of threads usage, i.e., the performance of the threads does not depend on the threads number (or, in other words, the execution time for the parallel part of the code is actually p times reduced when running on p threads).

Let us analyze the last condition in more detail.

2.2 Actual Efficiency of the Parallel Program

Let us inquire the issue: which maximum speedup can be achieved for the ideally parallel algorithm on some parallel computer cluster.

We consider implementation of DAXPY operation from BLAS1, which with the OpenMP directive can be written as following:

```
#pragma omp parallel for
for (i=0; i<n; i++) y[i] += a * x[i];
```

The numerical experiments were performed on the computer cluster [7] of the Institute of Numerical Mathematics of the Russian Academy of Sciences (INM RAS). The computer nodes specification from x6core queue that has been used for the experiments are:

- Compute Node Asus RS704D-E6;
- 12 cores (two 6-cores processors Intel Xeon X5650@2.67GHz);
- RAM memory: 24 GB;
- Disc memory: 280 GB;
- Operating system: SUSE Linux Enterprise Server 11 SP1 (x86_64).

Table 1. The efficiency of the DAXPY operation within OpenMP and MPI environment.

p	E^*_{omp}	S^*_{omp}	E^*_{mpi}	S^*_{mpi}
1	1.000	1.00	1.000	1.00
2	0.939	1.87	0.948	1.89
3	1.778	5.33	0.994	2.98
4	1.929	7.71	0.987	3.94
5	1.496	7.48	0.994	4.97
6	1.481	8.89	0.986	5.92
7	1.095	7.66	0.997	6.98
8	1.011	8.09	0.977	7.82
9	0.863	7.77	0.988	8.89
10	0.842	8.42	0.933	9.33
11	0.638	7.02	0.960	10.56
12	0.385	4.63	0.985	11.82

The Intel C compiler 4.0.1 with the MPI 5.0.3 support was used.

For comparison, the above mentioned code fragment has been run not only under the OpenMP environment but under MPI environment as well. The values of speedup (1) and the efficiency $E = S/p$ obtained are given in Table 1. Some specific results were defined for the parallelization by OpenMP:

- the expected reduction of the efficiency E^*_{omp} for large number of threads $p = 11, 12$ due to insufficient bandwidth of the memory channel;
- the unexpected superlinear speedup for $p = 3, ..., 8$ threads, that violate the Amdahl's law, probably due to coherent memory access operations and effective compiler processing.

In case of the MPI implementation the computational efficiency E^*_{mpi} for such an "ideal" algorithm has expectedly been very close to 1.

Thus, if we would like to improve the formula of the Amdahl's law (2) in accordance with the specific of the numerical experiment on the certain computer cluster, we should multiply the right-hand side of the Eq. (2) by E^*_{omp}:

$$S(p) = pE^*_{\text{omp}}/(1 + f(p-1)). \tag{3}$$

The value of E^*_{omp} here would be considered as given in tabular form in accordance with Table 1. Then the possible superlinearity would be included in formula (3), that prevents the failure of the Amdahl's law.

3 Parallel Computation Model for Distributed Memory Computers

The key peculiarity of the parallel algorithm execution on the distributed memory computer is a memory exchange operations and an additional loss of the

efficiency connected there with. Issues connected with the memory exchanges can be considered in more detail.

3.1 Message Transmission Rate

To estimate the time spent at the data exchanges the well-known formula can be used:

$$T_c = \tau_0 + \tau_c L_c, \tag{4}$$

where τ_0 is the initialization time for the message transmission, τ_c is the rate of the data exchange (i.e., measured by time of the data exchange of unit length), T_c is the time spent for the transmission of length L_c. Generally, the initialization time τ_0 (latency of the transmission) can be rather long, for example, $\tau_0 = 100\tau_c$, i.e., the time spent for transmission of 100 words can take just only two times more than the transmission of one word. However, if the length of the transmission is large enough, for example, greater than 1000, than the latency can be neglected.

The most effective algorithm implementation would be the implementation of a transmission of great length. Such an algorithms are called the algorithms with "large-grained" parallelism. If this algorithms class is analyzed, the simplified formula can be applied:

$$T_c = \tau_c L_c. \tag{5}$$

In other words, we neglect the latency of the communication network and consider that the rate of data transmission is specified only by network capacity. It should be noted that in the specified propositions the transmission time becomes linear with respect to the data length. Additionally, it means that the total length of all transmissions will define the total transmission time for several successive transmissions. Subsequently, this fact allows us to essentially simplify the efficiency estimation of the parallel algorithms analyzed.

3.2 Estimate of the Algorithm Parallel Efficiency

Let us introduce the same notations as in Subsect. 2.1. Let p be the number of processors used and $T(p)$ be the execution time for the algorithm on p processors. Respectively, the speedup that can be obtained by the algorithm will be expressed by the formula $S = T(1)/T(p)$, while the efficiency of the algorithm will be specified by the ratio $E = S/p$.

To estimate the computation time for the algorithm we need the knowledge of both characteristics of the analyzed algorithm and the parameters of the parallel computer used.

Let L_a be the total number of arithmetic operations of the algorithm and τ_a is the time spent per one such operation. Similar, let L_c be the total transmission length and τ_c be the time of transmission of the unit length. Then, the total time for arithmetic operations can be expressed by the formula $T_a = \tau_a L_a$ and the total time for communications is $T_c = \tau_c L_c$.

Now, everything is ready to speedup estimation, but we introduce two auxiliary values. The first one will describe the general characteristic of the parallel computer properties:

$$\tau = \tau_c/\tau_a, \tag{6}$$

specifying how many arithmetic operations can be performed when transmitting a number from one processor to another (in case of theoretically unlimited fast data transmissions or formally synchronous transmissions, $\tau = 0$; for computers with sufficiently fast transmissions we can expect approximately $\tau = 10$; while on case of slow communications we have about $\tau = 100$).

The second important value is the characteristic of the algorithm parallel properties:

$$L = L_c/L_a, \tag{7}$$

denoting a value being reverse to how many arithmetic operations are actually performed by the algorithm when transmitting a number.

Finally, we can estimate the speedup:

$$\begin{aligned} S = S(p) = T(1)/T(p) = T_a/(T_a/p + T_c/p) = pT_a/(T_a + T_c) = p/(1 + T_c/T_a) \\ = p/(1 + (\tau_c L_c)/(\tau_a L_a)) = p/(1 + \tau L), \end{aligned} \tag{8}$$

and, analogously, estimate the efficiency:

$$E = S/p = 1/(1 + \tau L). \tag{9}$$

As a result, we obtain a fairly simple formula for efficiency estimate, depending on two parameters τ and L only, characterizing parallel properties of computer and algorithm, respectively. At first glance, it is surprising that the last formula has no explicit dependence on the number of processors p, but what actually happens is that it implicitly presents in characteristic L via the dependence of all transmissions L_c total length with the given amount of processors p.

Let us summarize the assumptions that has been made during derivation of the upper bound of the speedup and efficiency of the parallel algorithm when running on the shared memory computer:

- in contrast to the Amdahl's law formula, it is considered that all computations are completely parallelizable and sequential part of the algorithm is absent ($f = 0$);
- the delay in computations is due to the data transmissions only, and the algorithms with synchronous communications are mainly suited for this model;
- parallel computations are well balanced, i.e., there is no delay due to imbalance;
- computational nodes are uniform, it means that parameter τ is the same for all nodes (though as it is known MPI can be performed on nonuniform computer systems);
- the arithmetic operations rate τ_a is independent on the number of processors p (for distributed memory computers it is performed more frequently, than on the shared memory computers with the use of OpenMP, see Table 1);
- the data transmission rate τ_c is independent on the number of processors p as well (this less obvious fact means the scalability of communication network).

3.3 Estimation of the Linear Algebra Algorithms Parallel Efficiency

Let us consider the application of the constructed speedup and efficiency estimates for some examples of linear algebra algorithms.

Example 1 (ideally parallel operations).

(a) Sum of two vectors:

$$Z_i = X_i + Y_i, \quad i = 1, ..., n. \tag{10}$$

(b) Vector normalization (multiplication by a constant):

$$X_i = \alpha X_i, \quad i = 1, ..., n. \tag{11}$$

(c) AXPY operation (as a combination of two above mentioned operations, intensively used in numerical methods, implemented in BLAS1):

$$Y_i = \alpha X_i + Y_i, \quad i = 1, ..., n. \tag{12}$$

(d) Multiplication of block-diagonal matrix by a vector, each block corresponds to certain processor, moreover a sparsity structure inside each block does not matter if total amount of nonzero elements inside blocks are about the same.

(e) Solution of linear system with block-triangular matrix when performing forward or backward substitutions. As in the previous case, the block structure can be arbitrary if the number of nonzero elements in each block triangle is about the same.

It is obvious that for these operations it is not necessary to perform the data transmissions ($L_c = 0$, and hence $L = 0$), therefore the speedup will be linear: $S = p$ for any value of τ, and the efficiency will be overall: $E = 1$. The computations are independent, and for cases (a)–(c) it is possible to exploit maximum number of processors $p = n$. It should be noted that in all cases the uniform load balancing is assumed, i.e., vector components amount for the cases (a)–(c) and number of nonzero elements inside the block for the cases (d) and (e).

Example 2 (dot product or inner product).

$$c = \sum_{i=1}^{n} X_i Y_i. \tag{13}$$

Firstly, we should locally compute the partial sum at each processor, and then it is necessary to compute the total summation and send the result to processors. By means of MPI library it can be done by using, for example, the function MPI_Allreduce(). The way of this function implementation is not fixed in MPI standard and is left at the discretion of specific MPI implementation. However, to estimate the speedup we can apply the simplest way by sending the partial

sums to a master processor and perform summation on it, and then distribute a result to other processors.

The total number of arithmetic operations (considering the summation with multiplication as a single operation, as well as a separate summation on the master processor) will be equal to $L_a = n + (p-1)$, but the total length of all data exchanges is $L_c = 2(p-1)$.

As while calculating L we are interested only in the ratio of these values, it is more convenient to write them down with respect to a local processor, i.e., $L_a = (n + (p-1))/p$ and $L_c = 2(p-1)/p$. Further, if not stated otherwise we will mean precisely such estimates.

As a result, the speedup estimate will look like:

$$L = L_c/L_a = 2(p-1)/(n + (p-1)), \tag{14}$$

$$S = p/(1 + 2(p-1)\tau/(n + (p-1))). \tag{15}$$

For example, for $n = 10^6$ and $\tau = 10$ we can calculate several estimate values:

$$S(p = 1) = 1, \quad S(p = 100) \approx 99.8, \quad S(p = 1000) \approx 980, \tag{16}$$

and for $\tau = 100$ with the same vector dimension we obtain:

$$S(p = 1) = 1, \quad S(p = 100) \approx 98, \quad S(p = 1000) \approx 800. \tag{17}$$

The dot product operation is very important and is frequently used in linear algebra. It is observed that in case the amount of processors increases up to $p = 1000$, there is a drastic fall of operation speed. The estimations provided by this paper indirectly confirm this observation.

Example 3 (multiplication of a dense matrix by a vector). Let us consider the matrix-by-vector multiplication for dense square matrix:

$$Y_i = \sum_{j=1}^{n} A_{ij} X_j, \quad i = 1, ..., n, \tag{18}$$

considering that on each processor the portion of block rows are stored, as well as the corresponding parts of vectors X and Y:

```
[:]   [== == ==]   [:]
---   ----------   ---
[:] = [== == ==] * [:]
---   ----------   ---
[:]   [== == ==]   [:]
```

To perform the multiplication, it is necessary to collect on each processor the copy of vector X of full dimension, and then to perform multiplication on the local part of the matrix A located on the processor.

Let n be a matrix dimension, and the matrix rows are distributed by processors equally, then:

$$L_a = n^2/p, \quad L_c = (n/p)(p-1), \quad L = L_c/L_a = (p-1)/n, \tag{19}$$

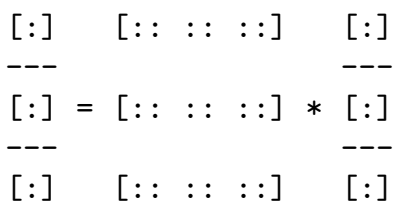
$$S = p/(1 + (p-1)\tau/n). \tag{20}$$

If $n = 1000$ and $\tau = 10$, then $S(p = 1) = 1$, $S(p = 10) \approx 9$, $S(p = 100) \approx 50$.

Example 4 (multiplication of a transposed dense matrix by a vector). Let us consider the matrix-by-vector multiplication for a transposed dense square matrix:

$$Y_i = \sum_{j=1}^{n} A_{ij}^T X_j, \quad i = 1, ..., n, \tag{21}$$

considering that on each processor the portion of block rows of A (block columns of A^T) is stored, as well as the corresponding parts of vectors X and Y:



To perform the multiplication, first, it is necessary to compute the local partial sum Z (of full dimension) as the product of the local block columns and the local part of the vector X, then send the parts of the vector Z to the respective processors, and, finally, sum the received parts of the vector Z to obtain the final local part of the vector Y.

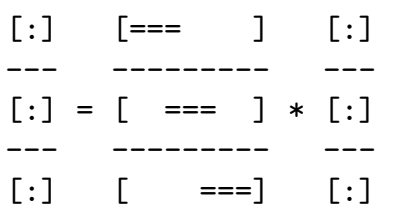
Let n be the matrix dimension, then:

$$L_a = n^2/p, \quad L_c = (n/p)(p-1), \quad L = L_c/L_a = (p-1)/n, \tag{22}$$

$$S = p/(1 + (p-1)\tau/n). \tag{23}$$

It is surprising, that, despite of the very different algorithm structure, the obtained estimate is the same as in Example 3. This is due to the same total length of interprocessor communications.

Example 5 (multiplication of a band matrix by a vector). Let us consider the matrix-by-vector multiplication for a band matrix stored by rows considering as it was stated before that each processor stores a portion of matrix rows as well as the corresponding parts of vector X and the resulting vector Y:



The portion of block rows of matrix A is stored on each processor, as well as the corresponding parts of vectors X and Y.

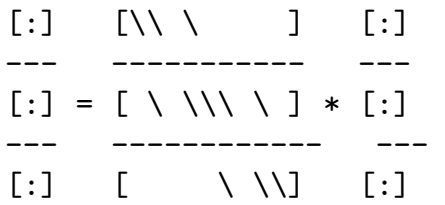
Let n be the dimension and r be the bandwidth of the matrix, then in order to perform the multiplication of the local part of the matrix each processor should additionally receive r components of vector X from two neighbouring processors:

$$L_a = (2r+1)n/p, \quad L_c = 2r(p-1)/p, \tag{24}$$

$$L = (2r/(2r+1))(p-1)/n \approx (p-1)/n, \quad S \approx p/(1+(p-1)\tau/n). \tag{25}$$

The most surprising in this estimate is the fact that it reproduces almost exactly the previous estimates and is almost independent on the half bandwidth r. It means, that although the number of arithmetic operation is reduced, the communication length is reduced in the same proportion.

Example 6 (multiplication of a sparse multi-diagonal matrix by a vector). Let us consider the matrix-by-vector multiplication for a sparse matrix with nonzero elements located on diagonals corresponding some discretization stencil. Let each processor stores a portion of matrix rows as well as the corresponding parts of vector X and the resulting vector Y:



Let n be the dimension and r be the semi-bandwidth of the matrix, and d be the total number of diagonals in the matrix (or number of vertices in the discretization stencil), then in order to perform the multiplication of the local part of the matrix A each processor (as in the previous example) should additionally receive r components of vector X from two neighbouring processors:

$$L_a = dn/p, \quad L_c = 2r(p-1)/p, \quad L = 2r(p-1)/(dn), \tag{26}$$

$$S = p/(1+2r(p-1)\tau/(dn)). \tag{27}$$

It worth to note, that for two-dimensional problem of size $n = m \times m$ with the use of 5-point discretization stencil the parameters of sparse matrix are equal to $r = m$ and $d = 5$. For three-dimensional problem of size $n = m \times m \times m$ with the use of 7-point discretization stencil we should take $r = m^2$ and $d = 7$.

It is worth to note, that the effective semi-bandwidth of the matrix depends on distribution of the domain to processors, for example, in three-dimensional case it is advantageous to cut the domain by 3D domains but not by slices. It may essentially reduce the total communication length and increase the efficiency of the sparse matrix-by-vector operation.

However, in comparison with the multiplication by a matrix with a dense band, the low efficiency of such an operation is due to the respectively less number of arithmetic operation for the same semi-bandwidth, and consequently for the same communication costs.

3.4 Conjugate Gradient Method

As the final example, we derive the estimate for the preconditioned conjugate gradient (PCG) method [8].

We consider the most simple but frequently used preconditioner: the block Jacobi structure with no overlap and incomplete Cholesky IC0 factorization of each block. The basic operations involved in this algorithm have already been studied in Subsect. 3.3:

- three "AXPY" operations (Example 1c);
- two inner "DOT" products (Example 2);
- multiplication of a sparse multi-diagonal matrix by a vector "MVM" (Example 6);
- solution of linear system with block-diagonal preconditioner matrix "SOL" (Example 1e).

We can write out now the speedup estimate for an iteration of PCG algorithm.

Example 7 (conjugate gradient method). The computational and communicational costs for a single iteration of conjugate gradient method with IC0 preconditioning consist of

$$\begin{aligned} L_a &= 3L_a^{\mathrm{AXPY}} + 2L_a^{\mathrm{DOT}} + L_a^{\mathrm{MVM}} + L_a^{\mathrm{SOL}} \\ &= 3(n/p) + 2(n/p) + (dn/p) + (dn/p) = (2d+5)n/p, \qquad (28) \\ L_c &= 3L_c^{\mathrm{AXPY}} + 2L_c^{\mathrm{DOT}} + L_c^{\mathrm{MVM}} + L_c^{\mathrm{SOL}} \\ &= 3 \cdot 0 + 2(2(p-1)/p) + (2r(p-1)/p) + 0 = (2r+4)(p-1)/p. \qquad (29) \end{aligned}$$

After that the "parallelism" characteristic of the algorithm can be expressed as

$$L = L_c/L_a = (2r+4)(p-1)/((2d+5)n), \qquad (30)$$

while the speedup estimation will be expressed as follows:

$$S = p/(1+\tau L) = p/(1+(2r+4)\tau(p-1)/((2d+5)n)). \qquad (31)$$

3.5 Numerical Experiment and Comparison with the Speedup Estimate

For the numerical experiments INM RAS cluster [7] with already described in Subsect. 2.2 computational nodes from queue "x6core" was used.

First, we compute the "parallelism" characteristic of the computer, which was applied in Subsect. 3.2 when deriving the estimate. Operation DOT over double precision vectors of length 10^6 was used to estimate the arithmetic performance of the cluster, while two simultaneous asynchronous data exchanges with the double precision vectors of the same length was used to estimate the transmission

rate. The communications were performed without overlapping with arithmetic operations. The following values were obtained:

$$\tau_a = 3.14 \cdot 10^{-10}, \quad \tau_c = 3.06 \cdot 10^{-8}. \tag{32}$$

It means that the main "parallelism" characteristic of the computer can be set to:

$$\tau = \tau_c / \tau_a = 100. \tag{33}$$

To verify the obtained estimates the developed in the INM RAS parallel program platform INMOST [9] was used. It can be loaded as a source code from [10]. As the model problem we have used the test program solver_test002 developed by the author of the paper, the program is accessible from the same site as well. The linear system matrix were constructed by discretization of 3D 7-point stencil for the domain of size $n = m \times m \times m$. The resulting linear system was solved by PCG method from the external package PETSc [11]. The additive Swartz method with no overlap and IC0 factorization in subdomains was used by setting the following parameters:

```
-ksp_type cg
-pc_type asm
-pc_asm_overlap 0
-sub_pc_type ilu
-sub_pc_factor_levels 0
```

A set of problems with different dimensions was considered, the dimension of domain in each direction was $m = 64, 96, 128, 160$. The total number of unknowns ranged from about 262 thousand to about 4 million, while a number of processors was chosen equal to $p = 1, 2, 4, 8, 16, 32, 64$.

For the final form of the PCG method speedup formula estimated by (31) the following parameters were used $r = m^2$, $n = m^3$, $d = 7$, and $\tau = 100$. For four considered linear systems the actual speedup with respect to the run on a single processor were obtained, and the plots of theoretical estimates by formula (31) were drawn as well. The obtained plots are presented on Figs. 1 and 2. It is worth to note that the plots behavior is qualitatively coincided.

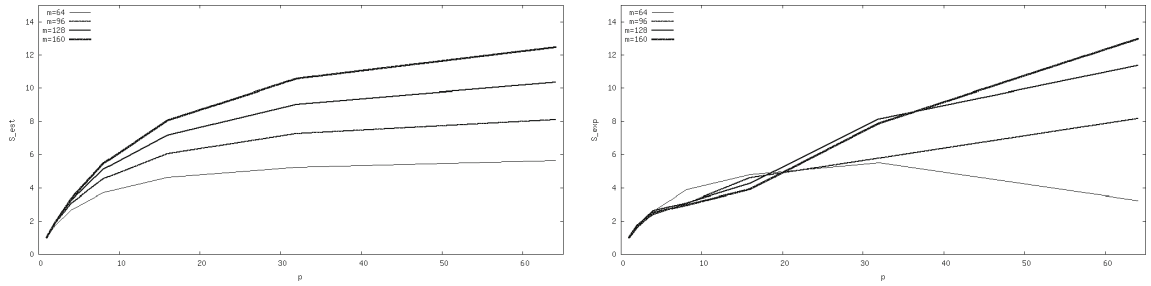


Fig. 1. The speedup estimated by formula (31) and the actual speedup for problems with $m = 64, 96, 128, 160$.

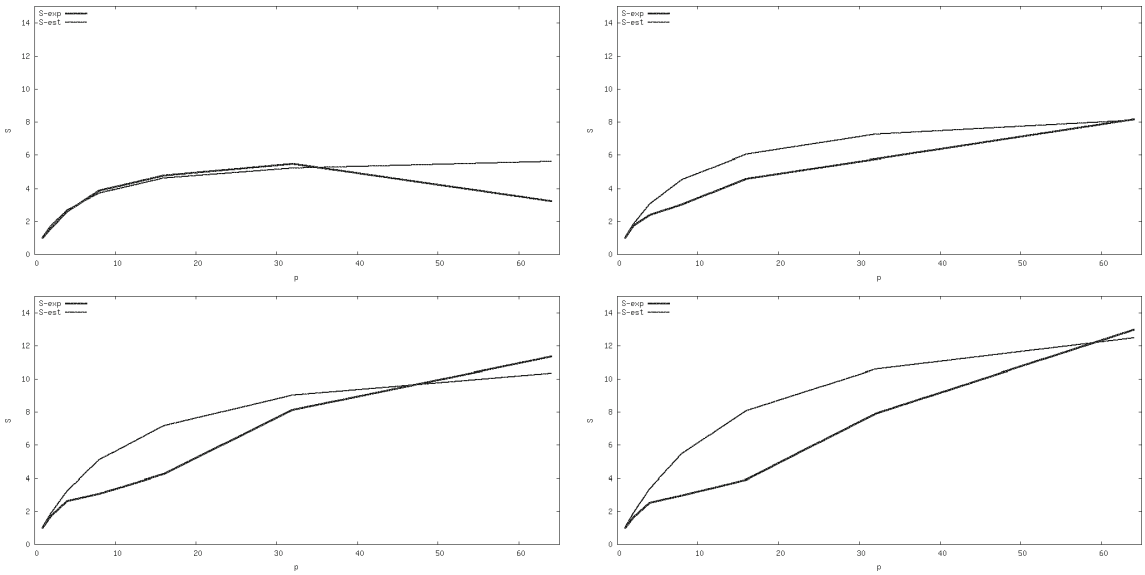


Fig. 2. Comparison of estimated and actual speedup for problems with $m = 64, 96, 128, 160$.

4 Conclusions

Two parallel computation models were presented for computers with both shared and distributed memory. Based on the macro-structure algorithm properties the speedup estimates were obtained for runs on parallel computers. The estimate for shared memory computers is built on the portion of serial computations of the algorithm, while the estimate for distributed memory computer clusters is based on the "parallelism" characteristics of both the considered algorithm and the computer in use.

The numerical experiments demonstrate that the theoretical speedup estimates and the actual experiment results are in qualitative agreement.

Acknowledgements. This work has been supported in part by RSF grant No. 14-11-00190.

References

1. Voevodin, V.V.: Parallel Computing. BHV-Petersburg, St. Petersburg (2002). (in Russian)
2. Bogachev, K.Y.: Parallel Programming. Binom, Moscow (2003). (in Russian)
3. Gergel, V.P., Strongin, R.G.: Parallel Computing for Multiprocessor Computers. NGU Publ., Nizhnij Novgorod (2003). (in Russian)
4. AlgoWiki: open encyclopedia of algorithm properties. http://algowiki-project.org. Accessed 15 June 2016
5. Amdahl, G.M.: Validity of the single-processor approach to achieving large scale computing capabilities. In: AFIPS Conference Proceedings, Atlantic City, NJ, 18–20 April, vol. 30, pp. 483–485. AFIPS Press, Reston (1967). http://www-inst.eecs.berkeley.edu/n252/paper/Amdahl.pdf. Accessed 15 June 2016

6. Antonov, A.: Under the Amdahl's law, No. 430. Computerra (2002)
7. INM RAS cluster. http://cluster2.inm.ras.ru. Accessed 15 June 2016 (in Russian)
8. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS, Boston (1996)
9. Vassilevski, Y., Konshin, I., Kopytov, G., Terekhov, K.: INMOST - A Software Platform and Graphical Environment for Development of Parallel Numerical Models on General Meshes. Moscow State University Publ., Moscow (2013). (in Russian)
10. INMOST - a toolkit for distributed mathematical modeling. http://www.inmost.org. Accessed 15 June 2016
11. PETSc (Portable, Extensible Toolkit for Scientific Computation). https://www.mcs.anl.gov/petsc. Accessed 15 June 2016