Topics in Parallel and Distributed Computing: Introducing Algorithms, Programming, and Performance within Undergraduate Curricula*†‡

Chapter 2 – What do we need to know about parallel algorithms and their efficient implementation?

Vladimir Voevodin[1], Alexande Antonov[2], and Vadim Voevodin[3]

[1]Lomonosov Moscow State University, Russia, voevodin@parallel.ru

[2]Lomonosov Moscow State University, Russia, asa@parallel.ru

[3]Lomonosov Moscow State University, Russia, vadim@parallel.ru

*How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms, Springer International Publishing, 2018, ISBN : 978-3-319-93108-1, Pages: 337.

†Free preprint version of this book: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cder_book_2

‡Free preprint version of volume one: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book

Abstract

The computing world is changing and all devices — from mobile phones and personal computers to high-performance supercomputers — are becoming parallel. At the same time, the efficient usage of all the opportunities offered by modern computing systems represents a global challenge. Using full potential of parallel computing systems and distributed computing resources requires new knowledge, skills and abilities, where one of the main roles belongs to understanding key properties of parallel algorithms. What are these properties? What should be discovered and expressed explicitly in existing algorithms when a new parallel architecture appears? How to ensure efficient implementation of an algorithm on a particular parallel computing platform? All these as well as many other issues are addressed in this chapter. The idea that we use in our educational practice is to split a description of an algorithm into two parts. The first part describes algorithms and their properties. The second part is dedicated to describing particular aspects of their implementation on various computing platforms. This division is made intentionally to highlight the machine-independent properties of algorithms and to describe them separately from a number of issues related to the subsequent stages of programming and executing the resulting programs.

Relevant core courses: Data Structures and Algorithms, Second Programming Course in the Introductory Sequence.

Relevant PDC topics: Parallel algorithms, computer architectures, parallel programming paradigms and notations, performance, efficiency, scalability, locality.

Learning outcomes: Faculty staff mastering the material in this chapter should be able to:

- Understand basic concepts of parallelism in algorithms and programs.
- Detect parallel (information) structure of algorithms.
- Understand deep relationship between properties of algorithms and features of computer architectures.

- Identify main features and properties of algorithms and programs affecting performance and scalability of applications.
- Use proper algorithms for different types of computer architectures.

Context for use: This chapter has to touch all the main areas of computer science and engineering: Architecture, Programming, Algorithms and Crosscutting topics. The primary area is Algorithms but these materials should be taught after learning the fundamentals of computer architecture and programming technologies. Materials of the chapter can be easily adapted for use in core, advanced or elective courses within bachelor's or master's curricula.

Introduction

Parallelism has been the "big thing" in the computing world in recent years. All devices run in parallel: supercomputers, clusters, servers, notebooks, tablets, smartphones... Even individual components are parallel: computing nodes can consist of several processors, processors have numerous cores, each core has several independent functional units that can be pipelined as well. All this hardware can work in parallel, provided that special software and the corresponding parallel algorithms are available.

After more than 60 years of development, a huge pool of software and algorithms has been accumulated for computers. The training process has been refined with the goal of learning programming technologies, and developing software, algorithms and methods to address various tasks. Now all of this is changing as the word "parallel" has literally found its way into everything: parallel programming technologies, parallel methods, parallel computing systems architecture, etc. Adding parallelism to existing training curricula definitely implies preserving the current serial programming methods, methodologies, technologies and algorithms, but many new things that never existed before need to be added [615126151261512]. How does one organize the parallel execution of a program to get a job completed faster? The question sounds simple, but answering it requires learning new ideas that have not been studied before.

In this chapter*, we present our experience in studying and teaching parallel methods of problem solving. This experience is based on using a large number of very different parallel computing systems: vector-pipeline, with shared and distributed memory, multi-core, computing systems with accelerators, and many others. Various forums for teaching parallel computing, parallel programming technologies, program and algorithm structures have been piloted at Lomonosov Moscow State University including general and special courses, seminars, practical computing exercises as part of the educational curricula at the Faculty of Computational Mathematics and Cybernetics, as

*The results were obtained in Lomonosov Moscow State University with the financial support of the Russian Science Foundation (agreement N 14-11-00190). The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

well as at the annual MSU Summer Supercomputing Academy [1]. Many of the ideas described in this chapter were implemented in the national project "Supercomputing education" [11211121].

The chapter consists of three sections. In the first section, we want to show, using numerous real-life examples, how many different properties of parallel algorithms and programs need to be taken into account to create efficient parallel applications. In the second section, these properties are described in a more systematic way, building on a structure that can be used to describe any algorithm. This helps to identify the most important properties for creating an efficient implementation. The description structure itself is universal, and not limited to any specific class of algorithms or methods. In the third section, we would like to show that the described materials can easily be incorporated into the educational process.

Before proceeding to the chapter we would like to make a special remark. This chapter is not a ready-to-use packaged lesson or a set of lessons, but rather ideas that should be presented throughout courses devoted to modern computational sciences and technologies. There is a high degree of freedom in choosing the methods for incorporating parallelism concepts into educational curricula, which do not require revolutionary changes and can be performed by existing academic staff within a current set of educational courses. We intentionally did not explain all the notions used in the chapter in a classical pedagogical way trying to concentrate on the main goal — to show a universal nature and wide use of parallel computing. From this point of view, our main target audience can be described as instructors who are already familiar with the subject and want to introduce parallel computing concepts into their courses, and parallel computing experts that teach related classes. At the same time it is really necessary to extend this audience involving a wide range of faculty staff into parallel computing as one of the most significant trends in computer science. The idea behind the chapter is to outline possible directions and ways how a teacher or instructor can incorporate parallel computing notions into any course.

2.1 What knowledge of algorithm properties is needed in practice?

In this section, we will consider several examples, focusing in each case on one property or another that determines how efficiently an algorithm is implemented. While reading the section,

it may seem that we are conflating parallel algorithms, parallel computing for different platforms, and performance issues. In a sense, this is true but this is necessary. If we are discussing high performance computing, we have to consider parallel algorithms, programming technologies, and architectures all together to ensure high efficiency of the resulting code.

By giving examples, we are not trying to explain every minute detail, give definitions, or explain newly introduced concepts, especially since many of them are quite intuitive. Our goals here are different. On the one hand, we want to show the great diversity of questions that arise in practice, the answers to which are determined by knowledge of the fundamental properties of algorithms and programs. On the other hand, by analyzing examples, we will gradually identify the set of properties that must be included in an algorithm description, and which teachers need to point out to their students.

2.1.1 Even in simple cases, it is important to understand the algorithm structure

Let's look at the classical algorithm for multiplying dense square matrices of size $n \times n$. Based on the formula

$$A_{ij} = \sum_k B_{ik} C_{kj},$$

it is quite natural to write the following version of the program (hereinafter in this paragraph matrix A is initialized with zeros):

```
for(i=0; i<n; ++i)
   for(j=0; j<n; ++j)
      for(k=0; k<n; ++k)
         A[i][j] += B[i][k] * C[k][j];
```

It has three nested loops and one assignment statement which calculates the element A_{ij}. The sequence of the loops in this example (i, j, k are the control parameters) is absolutely clear as it reflects the essence of the algorithm: for each element in matrix A (loop by i, loop by j), calculate the element A_{ij} (loop by k).

Let's perform a seemingly strange procedure: shuffle the three loops. We'll get a new fragment in which the loops can be organized in any of the six possible orders, for example (k, i, j) or (j, k, i). Will the new fragment provide the same results as the original program? A more general question also needs to be answered: "What loop order will provide the same result for the new program as the original version?" Below we show the two fragments mentioned above, with a loop order of (k, i, j) and (j, k, i); will the results of their execution be the same as those of the original fragment?

```
for(k=0; k<n; ++k)
   for(i=0; i<n; ++i)
      for(j=0; j<n; ++j)
         A[i][j] += B[i][k] * C[k][j];


for(j=0; j<n; ++j)
   for(k=0; k<n; ++k)
      for(i=0; i<n; ++i)
         A[i][j] += B[i][k] * C[k][j];
```

Questions like this may sound surprising, as there doesn't seem to be a reason why a formal loop interchange can result in a fragment equivalent to the original program. But the answer is even more surprising: in this example **any loop order** provides a result that is equal to the original fragment's results, accurate up to the rounding error. This begs two questions. Why does any loop order result in an equivalent fragment in this example? And the second question is why would we do something so strange as interchanging loops?

The first question can be answered by looking at the information structure of the matrix multiplication algorithm, shown in Fig. 2.1. The information structure is presented in a graph, where each vertex corresponds to one iteration of the three nested loops, and the vertices are connected with a directed edge [20222022] if one vertex calculates the data used in another one. We see n^2 independent computational branches, where each branch corresponds to the innermost k-loop for

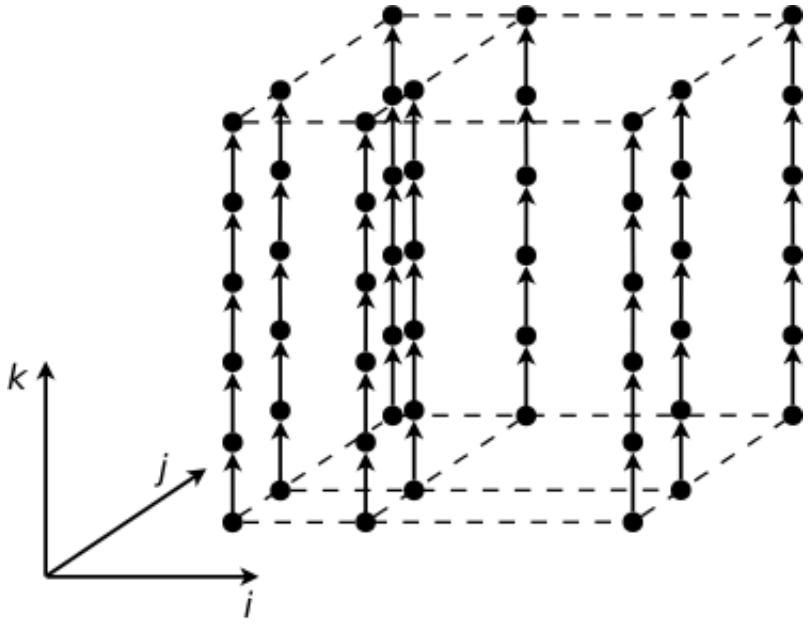


Figure (2.1) Information structure of the matrix multiplication algorithm

certain values of i and j, i.e. the calculation of the element A_{ij}. The picture is worth a thousand words. First, we see at once that all n^2 elements in the resulting matrix A_{ij} can be calculated independently from one another: the algorithm has a tremendous resource of parallelism, offering good prerequisites for writing a parallel program. Second, whatever the loop order in this fragment of the program: (k, i, j), (j, k, i) or any other, going through the vertices never violates the information relationship between the vertices; thus a fragment with any loop order is guaranteed to produce the same result as the original fragment. This feature of the information structure (information graph) for this algorithm explains the equivalence of the original and transformed fragments.

Now we only have to answer the second question: why did the need to interchange loops arise at all? Fig. 2.2 compares the execution time of the original program (with the i, j, k loop order) with fragments using other loop orders on different platforms. In some cases, a fragment with a loop order different from the classical (i, j, k) order works several times faster! By simply changing the loop order, we won't change the program result, but may significantly reduce its execution time. Why does this happen? This brings to the forefront another property that we need to study, assess and describe — the data locality within a program. In practice, both spatial and temporal locality can be of importance, so both types of locality are important in understanding the quality of an

algorithm's implementation. This is what we did for the example above: we revealed the parallel structure of the algorithm (fig. 2.1), understood its potential for equivalent transformations (six sequences of loops) and finally found fragments with the highest locality (i, k, j) or (k, i, j).

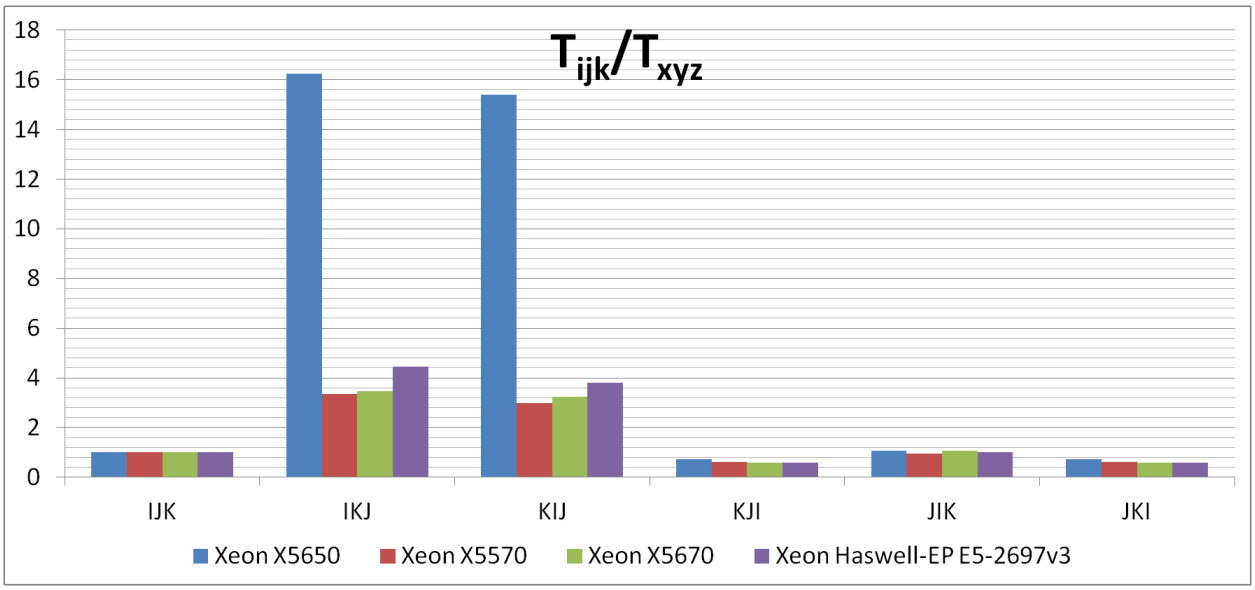


Figure (2.2) Comparison of execution times for matrix multiplication programs with various loop orders against the execution time for the classical order (i, j, k), the higher bars, the faster execution of the order (x, y, z)

This transformation of the program is quite simple, and therefore it is often performed by optimizers in modern compilers. At the same time, the compiler isn't actually guaranteed to perform the transformation. Moreover, even if the transformation is performed, there is no guarantee that the compiler will actually do it correctly and choose the one version with minimum execution time out of the six possible loop orders: we've already seen how complex the reasoning behind such a "simple" text change can be.

2.1.2 Simple properties can be very important, too

No detail is an afterthought in an algorithm description, and even seemingly obvious properties and parameters need to be properly observed. Let's look at the volume of input and output data for an algorithm. These figures almost invariably follow the algorithm formulation and are therefore considered obvious and shrugged off as being of little relevance. At the same time, these

figures aren't just important — they can completely determine the structure of the resulting program. Suppose we need to develop an algorithm to find the transitive closure of a directed graph. Let the graph consist of n vertices and m edges — these are the input parameters of the algorithm, which determine the input data volume. However, the output data volume for this task is strongly dependent not just on n and m, but also on the graph structure. In particular, if the input is a linear graph consisting of n vertices and $n-1$ edges, its transitive closure will contain $n(n-1)/2$ edges (see Fig. 2.3). This fact is no problem for processing relatively small graphs. However, it will be crucial for processing graphs representing social networks, which are obviously nonlinear but contain hundreds of millions of vertices and hundreds of billions of edges: due to immense volumes of output information (a quadratic dependence on the input data volume), the results will be impossible to store anywhere! The only way out of this situation is to restate the task so that it doesn't require listing every pair of vertices connected by edges. Input and output data volumes seem to be quite obvious parameters, but they do have to be thoroughly reviewed and described to understand the algorithm properties.

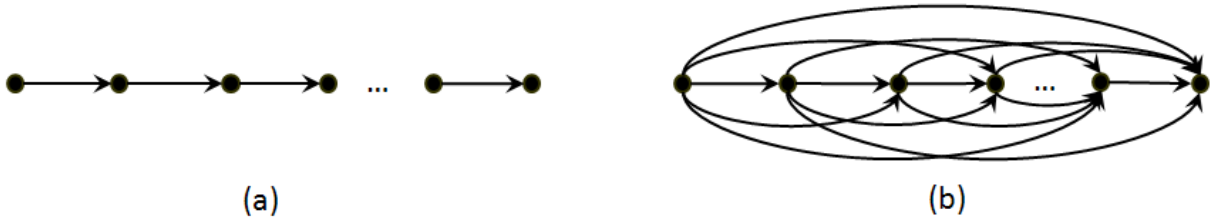


Figure (2.3) Linear graph (a) and its transitive closure (b): quadratic increase in the volume of output information

2.1.3 A new look at traditional concepts

There are other arguments in favor of considering every detail when describing algorithm properties. Let's look at an array of input data V and the total number of operations N in an algorithm. Both values are well known, each one is of interest in and of itself and is frequently used in practice. But it is equally important to pay attention to the ratio $P = N/V$. P stands for "computational intensity" and represents the number of operations per unit of input data required

to execute the algorithm.

Despite its simplicity, computational intensity is a very important feature of an algorithm. Suppose an algorithm's computational intensity is very high. This means input data requires a lot of processing before the algorithm's results can be obtained. As a result, this algorithm can be executed on an accelerator or a remote computing node, as the data transmission overhead will be low compared to the time it takes to process that data. This fact in particular explains why the Linpack test, with a computational intensity of n is so efficient on computers with distributed memory (about n^2 data elements are transferred to each computing node and about n^3 operations are performed on them). The same fact explains the low efficiency of an element-wise addition of two vectors using graphic accelerators: the time it takes to transfer $2n$ vector elements to a GPU completely offsets the rapid execution of n operations by the GPU.

In many cases determining the computational intensity requires taking output data into account, and not just input. In the vector addition example above, correctly evaluating the efficiency of the algorithm requires taking into account not just the time it takes to send input data to the GPU, but also the time it takes to get the results back. This will definitely reduce efficiency and decrease the computational intensity of this algorithm from $1/2$ to $1/3$, but that's the nature of the algorithm, and it must be considered. In practice, it is sometimes possible to increase the computational intensity by combining consecutive processing steps. For example, you might not want to return the results of a vector addition but instead continue processing at the accelerator, thereby eliminating unneeded data transfers.

2.1.4 Mathematics and parallelism

The information structure of an algorithm is an important concept, but it should not be used alone to evaluate the parallelism potential of an algorithmic approach. The math behind the algorithm plays an equally important role. Let's look at the classical vector elements addition algorithm:

```
s = 0;
for(i=0; i<n; ++i)
```

```
s += A[i];
```

The information graph for this algorithm is a linear graph (see Fig. 2.4a), where all vertices are connected with data dependency, which means only serial execution can be performed. Does this mean that neither the addition of vector elements, nor any other algorithms based on this operation can be used on parallel computers? Not exactly. The summation operation obeys the associative law, which allows us to tweak the original algorithm to achieve the appropriate degree of parallelism. Let's break all of the vector elements into non-overlapping groups, then find the subtotals for each group, and finally sum up the subtotals to get the total of all elements in a vector (see Fig. 2.4b). As the subtotals can be calculated independently (i.e. simultaneously), we now have a parallel version of the vector addition algorithm.

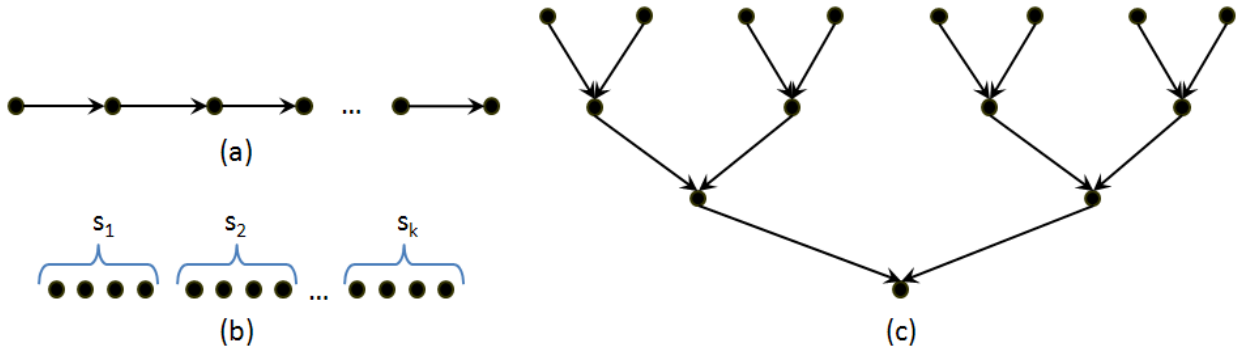


Figure (2.4) The information structure of several vector element summation algorithms: (a) classical algorithm, (b) parallel calculation of partial sums and (c) pairwise summation method

The associative law allows the elements to be added in any order with the same result, so other methods of implementation are also possible. Fig. 2.4c shows the information structure of the pairwise summation method, which is also a parallel modification of the original algorithm. All subtotals located at the same level can be calculated in parallel, moving between levels from top to bottom, until we get the desired result.

There are several important remarks regarding the serial-to-parallel process described above. First, it is math that enables us to make the key step: we would never get a parallel algorithm without using the associative law, based on just the knowledge of the information structure (Fig. 2.4a).

One needs to pay attention to possibilities like this when explaining or describing algorithm's properties, otherwise their potential will not be fully revealed. Second, the associative law implies that all operations are executed precisely, but computers operate using approximations of the original numbers. When we change the summation order, we may get a different result. Often the difference falls within the rounding error accuracy, which is negligible in most cases, but the fact that the associative law may not always work in computer arithmetic (just like the commutative and distributive laws) is something to be kept in mind. This explains, to a certain extent, the lack of reproducible results for parallel applications executed on supercomputers with a high degree of parallelism: literally every global MPI operation is based on the associative law, which results in various rounding errors and ultimately in different results when executing the same application. This is a serious issue that complicates the transition to Exaflop systems with an enormous degree of parallelism [105105]. Third, one needs to understand clearly that Figures 2.4a, 2.4b, and 2.4c represent **different algorithms**. The original task is the same: summing up the vector elements, but the algorithms are different. There are many differences between these algorithms: different information structures, different parallel complexity, different complexity of the respective programs, different rounding errors...

A situation like this, where knowing the mathematical basics of an approach can increase the degree of parallelism, is important in practice, and should be taken into account. Let's look at the task of finding the minimum spanning tree in a weighted graph G with E edges and V vertices. Suppose $MST(E)$ is the procedure for finding the minimum spanning tree. If we break the set of edges E into k non-overlapping subsets $E_1, E_2, \ldots E_k$:

$$E = E_1 \cup E_2 \cup \cdots \cup E_k,$$

then the basic MST procedure can be presented as follows:

$$MST(E) = MST(E_1 \cup E_2 \cup \cdots \cup E_k) = MST(MST(E_1) \cup MST(E_2) \cup \cdots \cup MST(E_k))$$

Minimum spanning trees for subgraphs with edges in the subsets E_1, E_2, $\ldots E_k$ can be found independently from one another; therefore, $MST(E_1)$, $MST(E_2)$, $\ldots$, $MST(E_k)$ procedures can also be performed in parallel. If we leave this mathematical fact aside, the algorithm's potential will not be utilized in full, as the available degree of parallelism grows with an increasing number of subsets E_i. After finding $MST(E_1), MST(E_2), \ldots, MST(E_k)$, it is necessary to join the minimum spanning trees found and perform the MST operation once again; however, the advantage of using parallel computing will still be substantial for $|E| >> |V|$.

This correlation does more than just increase resource of the parallelism. Its variations are exceptionally useful when processing very large graphs, as individual subsets E_i can be entirely stored and efficiently processed within RAM.

2.1.5 Parallelism can be inconvenient

If an algorithm has internal parallelism, this information is very important, but knowing this fact alone is not enough to make an efficient parallel program. Let's look at the example in figure 2.5a.

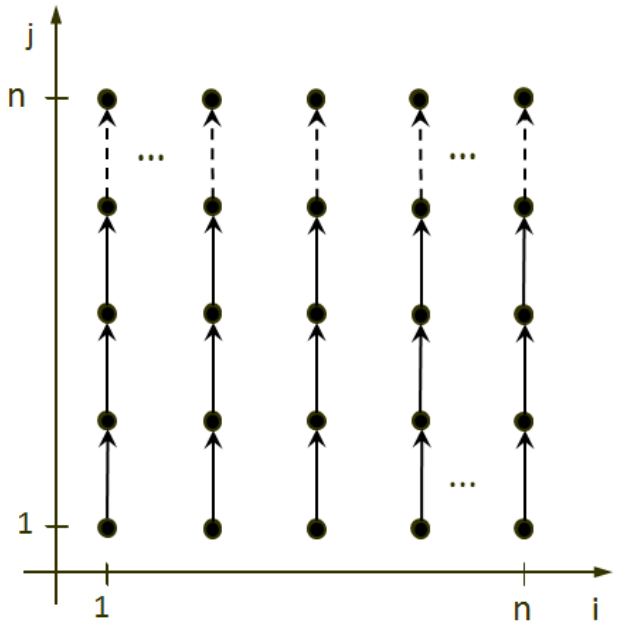
All iterations of the outer loop by parameter i are independent and can be executed in parallel (see Fig. 2.5b). To use the parallelism available in this fragment all we need to do is, for example, to place an OpenMP directive similar to "#pragma omp parallel for" — this makes the program parallel without any other modifications to its text. Moreover, all parallel branches will be perfectly balanced, as each one is used to execute n operations of the same kind.

The algorithm above has a very convenient structure and is easy to work with, but that's not always the case. Let's look at the example in Fig. 2.6a. The source code here looks very similar to the example we just looked at in Fig. 2.5a, but its information structure is completely different (see Fig. 2.6b).

None of the loops in the fragment 2.6a can be marked as parallel, since there are data dependencies in each dimension. But the fragment still has a great resource of parallelism; its serial complexity equals n^2 while the critical path of the information graph, reflecting the algorithm's parallel complexity, equals $2n-2$. To show the possibility of parallel execution of the algorithm,

```
for(i = 1; i <= n; ++i)
    for(j = 1; j <= n; ++j)
        A[i][j] = (A[i][j] * A[i][j-1]) / 2;
```

(a)



(b)

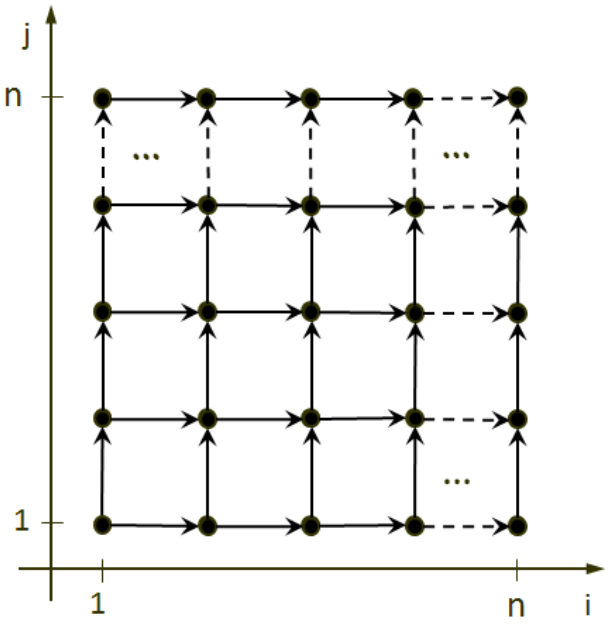
Figure (2.5) A fragment with “convenient” parallelism and its information graph

we can draw diagonals, as shown in Fig. 2.7a: all diagonals must be accessed serially, one after the other in ascending order, while all vertices located on the same diagonal can be computed in parallel (this type of parallelism is called skewed parallelism). To describe this execution method for the program, its text needs to be transformed; one possible transformation is shown in Fig. 2.7b. Moving from fragment 2.6a to fragment 2.7b is not a trivial task, as parallelism must be explicitly declared in most existing programming technologies.

The first reason why the parallelism in an algorithm can be called “inconvenient” is the need to transform the original code. Suppose the transformation has been completed; let’s go back to example 2.7. The first diagonal contains just one vertex, the second has two, the third — three; this number will increase to the value of n, then it will reverse its course, reaching 1 by the last diagonal. The available resource of parallelism change between steps, going from 1 to n and then returning to 1 again. It is extremely hard to develop an efficient way to execute this fragment: if too much computing resources (cores, processors, computing nodes) are allocated, some of them will be idle for a long time; but allocate too little resources — and the speed advantage will not

```
for(i = 1; i <= n; ++i)
    for(j = 1; j <= n; ++j)
        A[i][j] = A[i][j-1] * A[i-1][j];
```

(a)



(b)

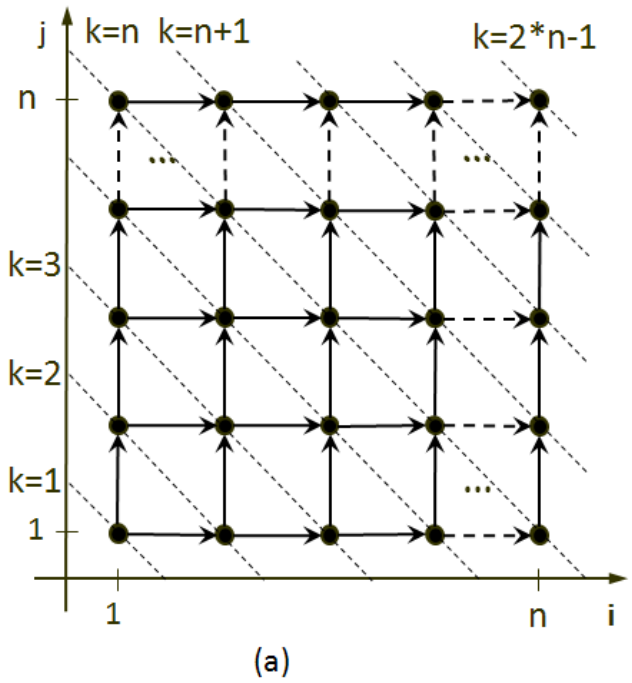
Figure (2.6) A fragment with “inconvenient” parallelism

be substantial compared to a serial implementation. This serious imbalance in computation is the second reason for the “inconvenience” of this type of parallelism.

How often do we focus on such properties of parallelism when we explain algorithm features during classes? In practice, the criterion of “convenient” or “inconvenient” parallelism in an algorithm is frequently the key factor in designing parallel applications.

2.1.6 It’s all about locality

Simple operations are efficiently implemented by a computer. This seems like an intuitively clear and correct thesis. It’s much more complex in practice. The simplicity of an operation is primarily understood as a simple algorithm structure, but when we talk about implementation efficiency, it is important to take into account not just the algorithm but also the program implementing it. Niklaus Wirth called one of his books “Algorithms + Data Structures = Programs” [23], and “Data Structures” are often what determines the efficiency of an algorithm’s implementation, even for very simple algorithms.




```
for(k = 1; k <= n; ++k)
   for(i = 1; i <= k; ++i)
   {
      j = k-i+1;
      A[i][j] = (A[i-1][j] * A[i][j-1]) / 2;
   }
for(k = n+1; k <= 2*n-1; ++k)
   for(i = k-n+1; i <= n; ++i)
   {
      j = k-i+1;
      A[i][j] = (A[i-1][j] * A[i][j-1]) / 2;
   }
```

(b)

Figure (2.7) Explicit identification of "inconvenient" parallelism

Let's look at several versions of a "triad", a basic operation used in many algorithms (see Fig. 2.8). Everything seems very clear and should not cause any efficiency implementation issues on modern processors: the structure is trivial, regular, no data dependencies, and the addition and multiplication operations are perfectly balanced. At the same time, the efficiency (the ratio of sustained performance to peak performance) for the simplest operation in Fig. 2.8a never exceeds 10% (see Fig. 2.9)! As we increase the number of input arrays (operations 2.8b and 2.8c)), the efficiency falls even further.

```
(a)    A[i] = B[i]*x + c;
(b)    A[i] = B[i]*x + C[i];
(c)    A[i] = B[i]*X[i] + C[i];
(d)    A[ind[i]] = B[ind[i]]*x+c;
(e)    A[ind[i]] = B[ind[i]]*x+C[ind[i]];
(f)    A[ind[i]] = B[ind[i]]*X[ind[i]]+C[ind[i]];
```

Figure (2.8) Versions of the "triad" operation

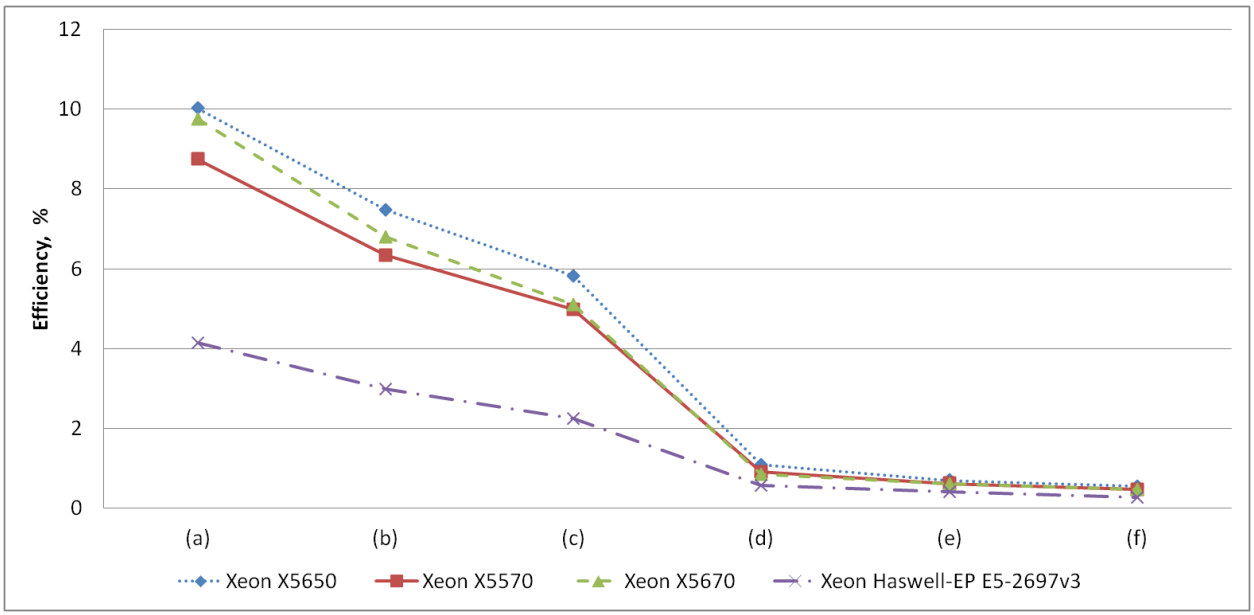


Figure (2.9) Efficiency (the ratio of sustained performance to peak performance) of different options for "triad" operation

What is the reason behind such low efficiency for a seemingly "perfect" operation? The main reason is the poor data locality in the resulting programs. The main data structure in every version 2.8a-2.8c is the arrays, and the elements in each operation are accessed serially, but each element is used only once. This means that spatial data locality is relatively low, and temporal locality does not exist at all. As the number of input arrays increases, the situation only gets worse. How often do we explain to the students what data locality is and how it affects program efficiency?

Let's go further and look at sparse data structures, instead of dense ones (operations 2.8d-2.8f). In this case we must use indirect addressing arrays (ind[i] array in Fig. 2.8), which further degrade the already low locality values and reduce efficiency to less than one percent! How often do we pay attention to this aspect of algorithms that operate with sparse arrays? One should not skip over a locality analysis when describing algorithm properties, otherwise the resulting program efficiency can be an unpleasant surprise.

2.2 Parallel algorithms: what to pay attention to?

In the previous section, we discussed just a few examples, but even this small amount of material shows how diverse are the algorithm properties that affect implementation efficiency. In this section we will focus on what needs to be included in training curricula, so as to draw attention to the nuances for efficiently implementing parallel algorithms for different parallel computing systems.

A description of any algorithm can reasonably be divided into two parts. The first part is dedicated to the theoretical properties of the algorithm, while the second describes the features of its implementation. This division is quite natural and helps to separate the machine-independent properties of the algorithm from the numerous issues that arise in practice. Both parts of the description are important: the first part helps to describe theoretical potential of the algorithms, while the second part shows the practical use of this potential. By learning information in the first part, students will understand the algorithm's general applicability, while the second part will help finding a way to efficiently implement it.

2.2.1 Parallel algorithms: theoretical potential

Let's look at an **algorithm's computational kernel**. It is the part of the algorithm that takes up most of the processing time. The computational kernel determines the quality of an algorithm's implementation in general, therefore it has to be an area of focus in the algorithm implementation process. If no acceptable implementation exists for an algorithm's computational kernel, it won't exist for the entire algorithm as well. Remember the analysis of the "triad" operation in the previous section (see Fig. 2.8): if we suppose it is the computational kernel of an algorithm, and the application uses sparse data structures, you can't expect high efficiency from the application in general (see Fig. 2.9).

The computational kernel does not have to be determined by operations on real numbers: addition, multiplication, division, square root, $\sin(x)$, $\cos(x)$, etc. For many algorithms, data load-/store operations, boolean or integer operations are the biggest bottlenecks. This doesn't change

anything, and a computational kernel consisting of such operations has to be singled out and described just as well. If most of the execution time for an algorithm is spent on matrix transposition, special attention must be paid to carefully storing and copying the data.

Another fact must be noted. Even though the full description of an algorithm can be quite large, the computational kernel is usually very compact, which allows the computational structure to be quickly understood and thus simplifies and speeds up the algorithm analysis.

Serial complexity, i.e. the number of operations that need to be executed in a serial implementation of the algorithm, is a highly important feature. Complexity is always expressed through parameters that determine the task size, and helps to quickly assess the viability of an algorithm's practical implementation. The operation type is not specified in any way, so whatever operations contribute the most to an algorithm's execution time shall be included in the formula for serial complexity. This can include operations on real numbers or integers, bitwise or memory loads/stores operations, array element updates, elementary functions, macro operations, etc. Arithmetic operations on real numbers prevail in LU decomposition, while large number factorization relies heavily on bitwise and boolean operations; this has to be reflected in the complexity evaluation.

If an algorithm has high complexity, then it must be used for large task sizes with extreme care. Moreover, if the algorithm is a component of another algorithm, then overall complexity can be even higher, and one should be even more careful. The computational complexity of a fast Fourier transform (Cooley-Tukey algorithm) for vectors with a length equal to powers of two equals $n \log_2 n$ complex addition operations and $(n \log_2 n)/2$ complex multiplication operations. At the same time, when looking at this algorithm, one should remember that a fast Fourier transform is often a basic component of other algorithms, being part of some large loops, which increases overall complexity.

All modern computers are parallel, so it is important to not just explain an algorithm, it is vital to simultaneously **show the algorithm's parallel structure**. This can be done, for example, with the help of an information graph, sometimes called an algorithm graph, data dependency graph or data-flow graph. Determining the parallel structure of an algorithm is always the first step in creating a parallel program, regardless of what specific parallel computing system the program is

being written for. This step is very important, and if the algorithm's parallel structure is known (see Fig. 2.1, 2.5b, 2.7a), many subsequent decisions become obvious.

There are many possible options for representing the information structure of an algorithm. For some algorithms, the information structure must be shown in every detail; for others a macro structure is more important. A lot of information is available in various forms of information graph projections, which clarify the algorithm's regular components while hiding insignificant details. Sometimes it may be useful to show a pattern in the graph that changes with the values of external variables (e.g., matrix sizes): we often expect "similar" behavior in the information graph, but it isn't always obvious in practice.

Visualization of the information graph can be very useful for studying various algorithm properties. But the task of displaying an information graph is not trivial. To begin with, the information graph can potentially be endless, as the number of vertices and edges is determined by external variables which can be very large. In this situation it helps to look at likenesses, as described above, which consider graphs for different values of external variables as "similar": it is almost always enough to present one small graph, stating that graphs for other values will look "exactly the same." Not everything is so simple in practice, however; and one should be very careful here.

Next, an information graph is potentially a multi-dimensional object. The most natural coordinate system for placing vertices and edges in an information graph relies on the nested loops in an algorithm's implementation. If nested loops have not more than three levels (see Fig. 2.1), the graph can be placed in the traditional three-dimensional space, but complex loop constructs with nesting levels of 4 or more require special methods for presenting and displaying the graph.

There are many difficulties here, but also many ways to deliver the information to the students. The main task is to show the information structure of an algorithm so as to demonstrate all its key features, its parallel structure features, edge sets features, regularity areas and, vice versa, areas with an indeterministic structure dependent on the input data, etc. Teachers very rarely talk about parallel algorithm structure, while this is required in practice more and more often.

After telling about the parallel algorithm structure, one should proceed with **describing its resource of parallelism**. The main characteristic is **parallel complexity**, which is understood

as the number of steps needed to execute this algorithm given an infinite number of processors (functional units, computing nodes, cores...). The concept of infinite parallelism is somewhat idealistic, but it helps to understand the advantages offered by the parallel execution of an algorithm. Parallel complexity of the fast Fourier transform (Cooley-Tukey algorithm) mentioned above for vectors with lengths equal to a power of two is $\log_2 n$, which means this algorithm can potentially be executed $1.5n$ times faster.

The concepts of a canonical parallel form or the critical path of an information graph are often used to evaluate and describe the resource of parallelism. The height of the parallel form, or the length of the critical path, determine the algorithm's parallel complexity, while the level width is determined by the number of processors needed at a specific level; all of this can be used to describe algorithm's properties. The complexity of an algorithm featuring the summation of n vector elements is reduced from $n-1$ in a serial implementation to $\log_2 n$ in the parallel version, with the number of operations executed in parallel at each level (i.e. at each step of the pairwise parallel algorithm) falling from $n/2$ to 1.

Parallelism in an algorithm often has a natural hierarchical structure. This fact is very useful in practice and should be reflected in a description of algorithm's properties. This hierarchical parallelism structure is well reflected through a loop profile of the resulting program (including, in general, the call graph). Fig. 2.10 shows a loop profile of a program, where each square bracket corresponds to a one loop of the program and nesting structure of the brackets repeats nesting structure of loops. Information that the outer loop (marked by '1') and all inner loops (marked by '2') are parallel (their iterations are independent) substantially improves the perception of the original algorithm's structure.

Figure (2.10) Resource of parallelism of a program: parallel loops are marked by '1' or '2' in the loop profile of the program

When explaining algorithms, it is important to pay attention to the **algorithm properties**

which can prove important during implementation. We mentioned some of them in the previous section, namely **computational complexity** or **input/output data volume**: these properties are simple, but they often determine the quality of the future implementation.

Application efficiency and the **balance of the computation process** are two closely related concepts. The main challenge is that the balance can appear in different ways. This can include balancing between different types of operations, particularly between arithmetic operations (addition and multiplication) or between arithmetic operations and memory access operations. This can also include computational balance between different parallel branches of the algorithm. On the one hand, load balancing is a necessary condition for efficiency of a parallel algorithm. At the same time, this is a very challenging task, and one must explicitly show how many of these features the algorithm has. If ensuring of balance is not obvious, it is recommended to describe possible ways for solving a task.

An important aspect in practice is the **determinacy of an algorithm**, which can be understood as the consistency of the computational process structure. From this viewpoint, the classical multiplication of dense matrices is a highly deterministic algorithm, as its structure, given a fixed matrix size, does not depend on the input matrix elements. Multiplying a sparse matrix by a vector, when the matrix is stored in a special format, is no longer deterministic: data locality depends on the structure of the input matrices. An iterative algorithm with precision-based exit is also not a highly deterministic one: the number of iterations, and therefore the number of operations, changes depending on the input data.

The reason for pointing out determinacy as a property is clear: working with a deterministic algorithm is easier, since a structure, once found, will determine its implementation quality at all times. If determinacy is missing, this should be specially pointed out, along with a description of how indeterminacy affects the structure of the computational process.

A serious reason for the indeterminacy of a parallel program is a change in the execution order of associative operations. A typical example is the use of collective MPI operations by a group of parallel processes, such as summing elements of a distributed array. The MPI runtime system chooses the operation execution order assuming compliance with the associative law; rounding

errors change for each program run, introducing changes in the program output. This is a serious issue often encountered on systems with massive parallelism, and it affects the reproducibility of results of parallel programs. If analysis of an algorithm's structure shows that a parallel application cannot work without collective operations, this property must also be kept in mind.

Interestingly, in some cases, determinacy can be "enforced" in an algorithm by introducing macro operations, which makes the structure not only deterministic but also more clearly understandable.

An important aspect is a **description of bit capacity needed to execute the algorithm's operations** (precision). In practice, executing all arithmetic operations on real numbers with double precision is rarely required, as this doesn't affect the algorithm's stability or the accuracy of the output. If most operations can be performed using a float type, and just a few fragments need to be changed to double, this fact must also be explicitly mentioned, as it can substantially improve implementation efficiency.

2.2.2 Parallel algorithms: implementation features

In the beginning of this section we discussed two parts of the description for any algorithm: a description of its theoretical potential and its implementation features. The properties considered above are related to the first part of the description. This information is important and relevant, it has to be explained, but it is just as important to look ahead and point out some possible stumbling blocks that can be encountered in the process of implementation. This is what we will address below.

The issues of data locality and computation locality are rarely included in any training courses, but locality has a very high impact on the efficiency of program execution on modern computing platforms. To get the whole picture of an algorithm's implementation features, it is important to analyze both temporal and spatial locality, noting positive and negative factors related to locality, and under which conditions and situations they are caused. It is important to mention how locality changes when moving from a serial to a parallel implementation, and to highlight typical memory access patterns for a program implementing the given algorithm. We should also

mention the potential correlation between the available programming language constructs and the locality exhibited by the resulting programs.

It is useful to show memory access profiles for computational kernels, which often explain the efficiency of the entire application. Fig. 2.11 shows memory access profiles for programs that implement FFT, LU decomposition, dense matrix multiplication and random memory access. Each small red dot corresponds to one memory access operation. The X-axis shows the serial numbers of memory access operations arranged in the order they were performed during the program execution. It is similar to the timeline chart, but in this case we analyze only the order of memory accesses, not the particular time when they were performed. The Y-axis indicates the memory address used in each particular memory access operation.

For example, figures 2.11d and 2.11e show the memory access profiles for two versions of matrix multiplication algorithms, which makes it clear why the (IKJ) fragment is executed much faster than the (JKI) fragment: the profile in Fig. 2.11d has a much higher spatial and temporal locality than that in Fig. 2.11e.

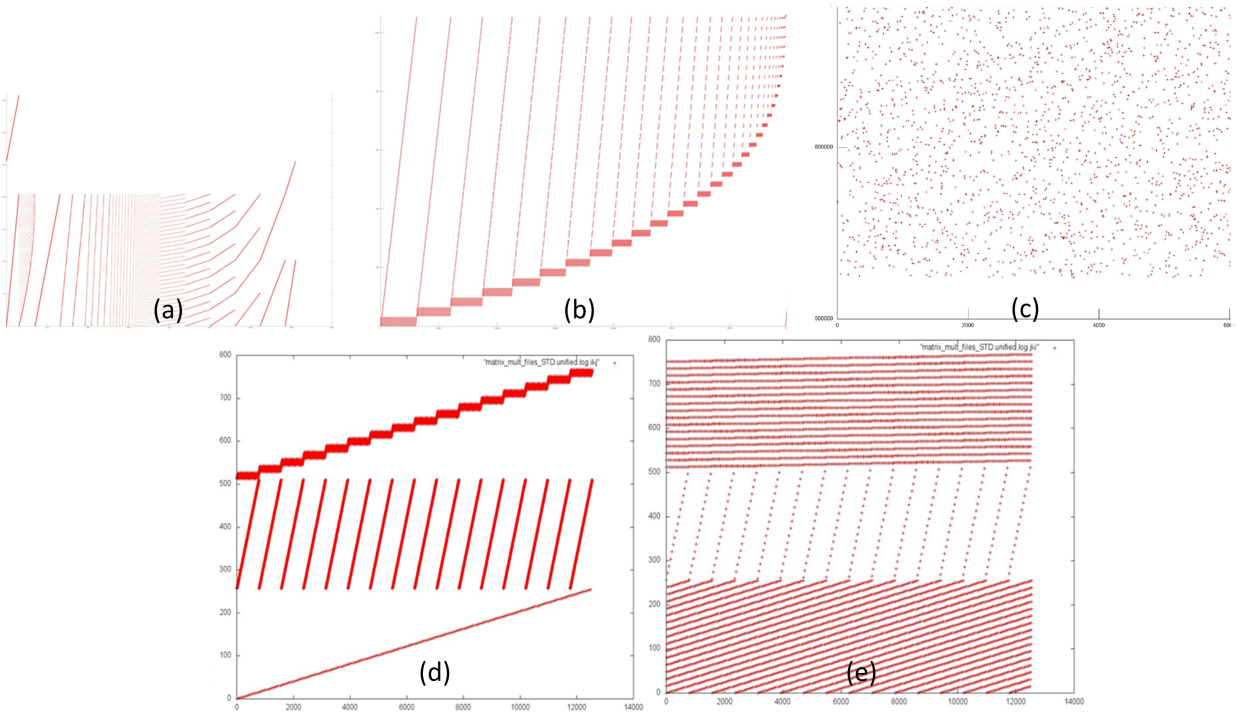


Figure (2.11) Memory access profiles for programs implementing FFT (a), LU decomposition (b), random memory access (c) and dense matrix multiplication with different loop orders (d,e)

Knowing the algorithm's resource of parallelism, it is important to show the **opportunities for equivalent transformation of the programs**: the students will see the program features for computers with certain architectures, and will feel freedom in transforming the programs and obtaining program optimization skills. Let's go back to the program with loop profile shown in Fig. 2.10. It was mentioned that the outer loop (marked by '1') has many iterations, while the inner loops were very short. In this form, the parallel structure is suitable for SMP computers, allowing parallel execution of the outer loop iterations by processor cores. However, if the target system is a vector-pipeline computer, the efficiency will be low for sure: only the short innermost loops (marked by '2') can be vectorized. As a way out of this situation, we can perform a series of elementary transformations on the loops (see Fig. 2.12), moving the long outside loop (marked by '1') inside. The transformation is certainly not trivial, and the program must effectively be "turned inside out." During some transformations we obtain loops which are not perfectly nested ("dots" in the fourth and fifth loop profiles denote additional statements "between" loops), but all the transformations are fully equivalent (program's information structure remains intact, and we operate strictly within the available resource of parallelism). If you know about this freedom in code transformations, you can easily compose a variant of code which matches well any target architecture.

It is worth noting that such non-trivial transformations can't always be automatically identified and performed by the compiler, which means that the programmer himself must be aware of such features.

Scalability is one of the central notions in parallel computing, which shows how efficiently the algorithm and the program implementing it can use the available processing cores, processors and computing nodes. This is an important idea since all computers are parallel today, yet it is a highly complex one. Application's scalability potential is originally determined by the algorithm, but can be reduced substantially depending on the programming technology, bad data distribution, inadequate composition of the parallel program, and many other reasons.

Many things can be discussed with students here: strong scalability, weak scalability, wide scaling, possible reasons for low scalability. It is interesting to compare scalability of different algorithms that address the same task. It is important to show the connection between algorithm

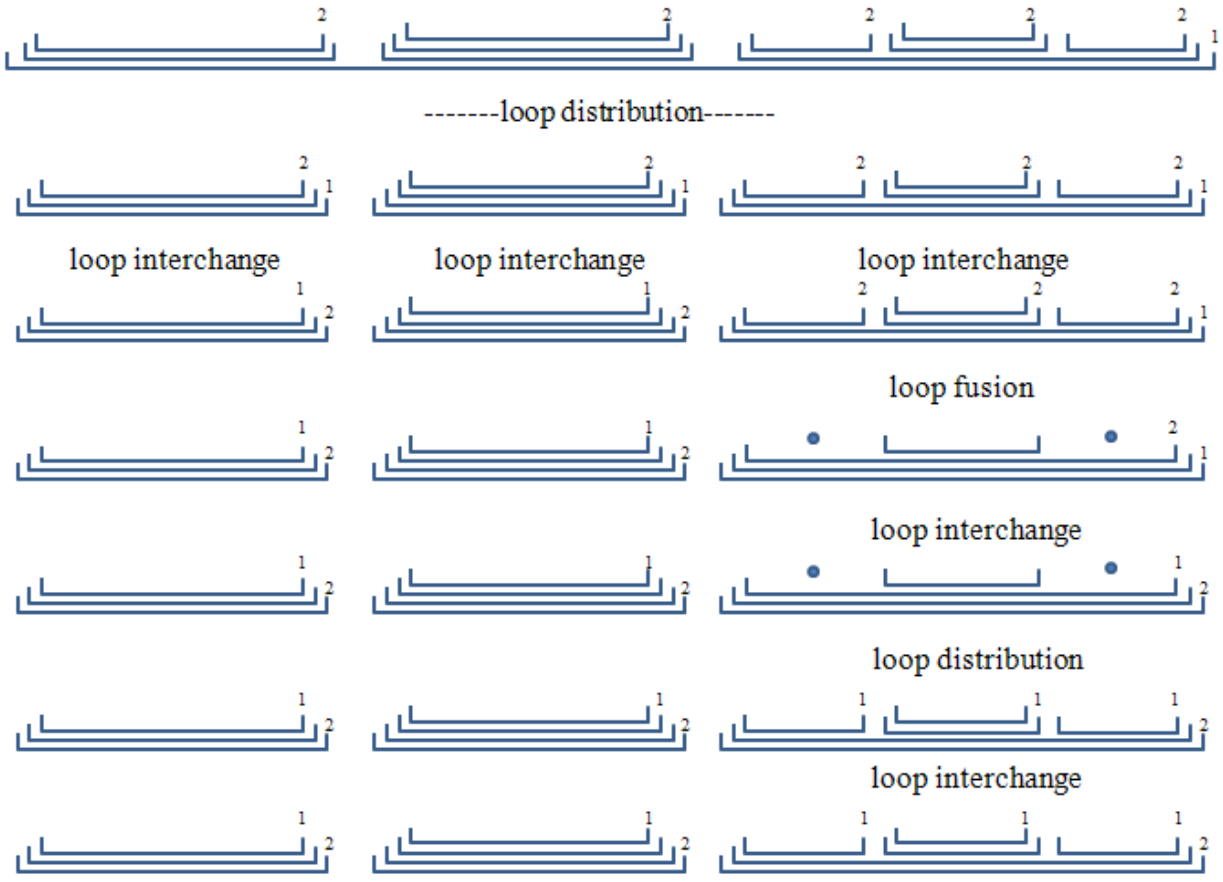


Figure (2.12) Series of loop transformations that convert the outermost loop '1' in the original fragment into the innermost loop

properties, program structure and computer architecture features, which lays the groundwork for co-design technologies and determines the scalability of parallel applications.

When explaining this idea, the most efficient argument is the behavior of the actual scalability of a given algorithm's various implementations, depending on the number of processors and the problem size. An important thing here is to find the right correlation between the number of processors used and the problem size to highlight all points of interest in the behavior of a parallel program, such as achieving maximum performance, and the more subtle issues that arise, for example, out of the algorithm's block structure or memory hierarchy.

Fig. 2.13a shows scalability for an MPI implementation of the classical dense matrix multiplication algorithm depending on the number of processes and the size of the dataset. The chart clearly displays areas with higher performance corresponding to different levels of cache memory.

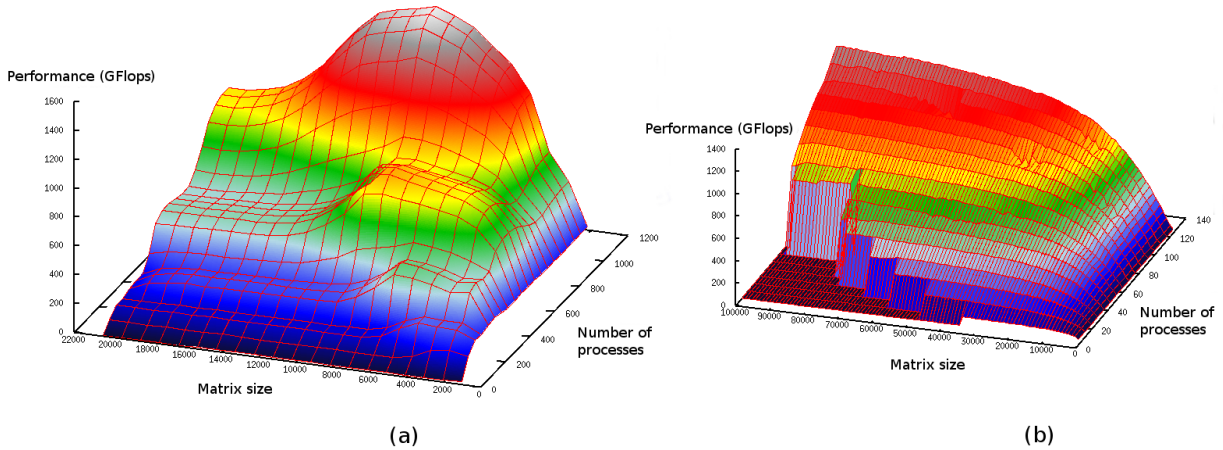


Figure (2.13) Scalability of the MPI implementation of the dense matrix multiplication algorithm (a) and Linpack benchmark (b), MSU “Lomonosov” supercomputer

Fig. 2.13b shows good scalability for the Linpack benchmark: as the number of processors grows, so does the performance for any matrix size shown in the chart. Some values are missing from the front part of Fig. 2.13b, as large tasks cannot fit into the memory of a small number of processors.

In addition to scalability, **performance and efficiency** are other concepts of particular interest in understanding the quality of a parallel algorithm implementation. These two notions are closely related and are often viewed not in abstract terms but in combination with a specific computing system. Efficiency can be understood differently: as parallel efficiency, or as efficiency compared to peak performance indicators for the computing system.

Algorithms or their implementations can possess specific features that prevent performance and efficiency from exceeding certain limits. These features need to be singled out and discussed with students, as they will likely run into something similar in practice. Fig. 2.14 shows the performance and efficiency for Poisson’s equation solution using the discrete Fourier transform depending on the number of processors and the problem size. Despite using serious computing resources (the experiments were conducted on the “Lomonosov” supercomputer [16] at Lomonosov Moscow State University), the program’s performance was very low, with efficiency never exceeding 1.5% and falling quickly as the number of processors grew (the main reason of poor efficiency is low data locality). These facts need to be mentioned if we want students to have a realistic perception

of what modern parallel computing systems are capable of, to understand peak performance figures correctly and to clearly understand the reasons why these situations arise.

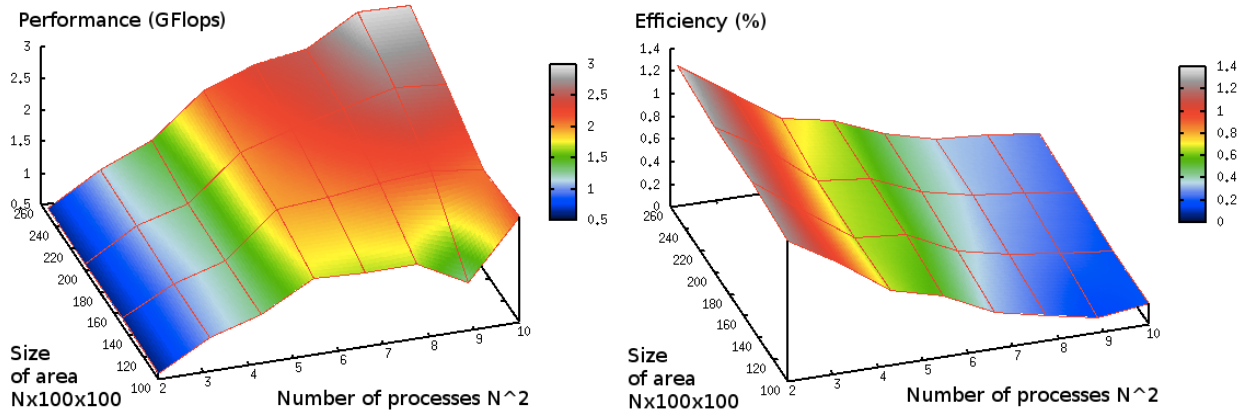


Figure (2.14) Performance and efficiency of Poisson's equation solution using the discrete Fourier transform

Another subject for a more professional and detailed discussion is the search for the root causes for low performance and efficiency in parallel applications. The task is not a simple one. There isn't currently a single, universally accepted methodology or the respective tools to conduct such analysis. In fact, this requires conducting a comprehensive analysis adequate for modern supercomputer co-design technologies. One would need to assess and describe the efficiency of memory access operations, the efficiency of using the resource of parallelism inherent in an algorithm, the efficiency of using communication networks and particular features of the communication profile, the efficiency of input/output operations, and many other aspects. Sometimes average aggregated characteristics of a program are generally sufficient; in some cases, it is necessary to show lower-level monitoring data such as CPU load, cache misses, InfiniBand network usage intensity, etc. Tools like TAU [18], Scalasca [17], Vampir [19], and JobDigest [2] provide a good understanding of the parallel program quality, and students need to be taught how to use them. Moreover, using such quality maintenance tools for parallel applications must become an integral element of the software development cycle.

While discussing the properties of a parallel algorithm, it is also important to evaluate its potential for the specific architecture of various classes of computing systems, and the specific

parallel programming technologies used. The computational kernel is compact and has high computational intensity — a good condition for using accelerators. The interaction between parallel processes is intensive and results in major overhead on delivering messages — this means that developers should look at computers with shared memory or use one-sided data transfer functions like Put/Get. An algorithm has SIMD parallelism, so graphics processors can efficiently implement it. By adding these touches to the description of algorithm's properties, we pursue the main goal of showing students a strong connection between the parallel algorithm properties and the computer architecture features, which form the grounds for creating high-quality parallel applications to efficiently solve real-life problems.

Our experience in analyzing algorithm properties and implementations not only determined the key issues that are addressed in this work, but also became the foundation for the AlgoWiki project [4343]. The project's main goal is to provide a description of the fundamental algorithm properties that enable full understanding of their theoretical potential and their implementation features regarding various classes of parallel computing systems. The project is expected to result in the development of an open encyclopedia based on wiki technologies and available to the entire academic and educational community. The first version of the encyclopedia is available at [13], where users can describe both their own pedagogical experience and their knowledge of specific parallel algorithms. The encyclopedia already contains many useful and live examples that can be used in lectures and seminars to explain the particular features of parallel algorithms.

2.3 How does one make a training curriculum parallel?

Parallel algorithms, parallel computing systems, parallelism — all of these are fundamental concepts that must be at the foundation of any curriculum on computational mathematics, applied mathematics, and generally of any curriculum on Computer Science. We are implementing this approach at the Faculty of Computational Mathematics and Cybernetics at the Lomonosov Moscow State University (CMC MSU), starting with Bachelor's degree coursework and continuing with many Master's degree and post-graduate education programs. In this section we would like to show the great diversity of methods and techniques that can be used in the educational process to

support parallel computing topics in various training programs. The choice of particular materials needed in each specific case is up to teachers. Our goal is to help selecting the most suitable methods for including this topic in a curriculum, to inspire the teacher to explore various directions, and to suggest potential pedagogical techniques for implementing this in practice.

2.3.1 Parallelism concepts — in every lecture course

We analyzed the structure of lecture courses in the Bachelor's degree programs at CMC MSU, and it turned out that parallelism concepts can be added to each one rather easily and gracefully. Some courses only require parallelism to be mentioned; for some, examples would need to be replaced with those having parallel specifics, but without any impact on the logical flow of presentation; sometimes, a course would need to add new lectures. However, this modification, turning the classical "serial" curriculum into a parallel one based on the ideas of parallel computing, does not present any specific challenges. Let's give some examples of how individual training courses can be modified; this will make it clearer regarding how parallelism can be introduced into any training course.

Algorithms and algorithmic languages (1st year). The concept of parallel execution for an algorithm needs to be introduced right in the very first semester of study (at an intuitive level).

Linear algebra and analytical geometry (1st year). For simple examples (summing vector elements, dot product), introduce the idea of computational complexity, discuss the possibility of parallel execution (without any theoretical justification), and perform assessments of parallel execution timing. The information structure of a classical matrix multiplication algorithm should be shown, pointing out the possible options for parallel execution.

Computer architecture (1st year). Introduce the principles of pipeline and parallel data processing, explain the architecture of multi-core processors, introduce the notions of independent functional units, vectorization, peak and sustained performance. Explain organization of memory subsystems in modern processors, and memory hierarchies.

Physical basics of designing computers (2nd year). Computer representations of numbers, accuracy, rounding, parallel bitwise execution of arithmetic operations. Rounding, the laws of

exact and machine arithmetic.

Operating systems (2nd year). Parallel processes, the fork/join model, synchronization methods, process interaction methods, deadlocks, determinacy, correctness, viability, fault tolerance.

Discrete mathematics, graph theory (2nd year). Two-dimensional grid, n-dimensional torus, n-dimensional hypercube, etc. as examples of multi-processor system topologies. The shortest path between nodes, critical path length, the outcome degree of graph vertices, routing and fault-tolerance. The need for and complexities of processing ultra-large graphs.

Algorithm complexity (3rd year). The serial and parallel complexity of an algorithm, computational intensity of an algorithm.

Introduction to numerical methods (3rd year). Information structure of implicit and explicit numerical methods, the trade-offs between computational complexity and the parallel structure of algorithms (explicit methods have good parallel structure but can have slow convergence, while implicit methods are serial in nature but can have better convergence).

Mathematical physics equations (3rd year). Parallel problem-solving methods, key steps in problem-solving on computers: task — algorithm — programming technology — program — compilation — execution, the need to preserve parallelism at every problem-solving step.

Databases (4th year). Parallel DBMS, ultra-large DBMS, load balancing in parallel query execution.

Optimization methods (4th year). Examples of parallel optimization methods.

Distributed systems (4th year). Metacomputing concepts, grid, cloud technologies, distributed processing technologies.

A detailed description can be provided for each training course, but this is not in the scope of this work. The idea is to show how naturally the ideas of parallelism can be included into almost every lecture course in a bachelor's degree program.

2.3.2 Emphasis on parallelism in exam and test questions

Questions on exams and tests for various disciplines must be formulated in such a way as to explicitly mention the ideas of parallelism, rather than conceal them. Accents in the questions can vary and they can change a flavour of the questions — several possible question formulations are shown below, which could be used, with minor modifications, for exams within many disciplines.

1. Types of parallel data processing, their features.
2. Evaluating the computational complexity of large tasks.
3. Memory hierarchy, computational locality, data locality.
4. Amdahl's law, its corollaries, and superlinear speedup.
5. Quality indicators of parallel programs: speedup, efficiency, scalability.
6. Strong scalability, wide scaling, weak scalability. Isoefficiency.
7. Parallel implementation of problems characterized by high computational complexity with matrix multiplication as an example.
8. Graph models of programs, their relationship.
9. The concepts of information dependency and information independence. An algorithm's resource of parallelism.
10. Information structure of algorithms. The critical path of an information graph.
11. Equivalent transformations of programs. Elementary loop transformations.
12. Types of parallelism: finite parallelism, massive parallelism, coordinate parallelism, skewed parallelism.
13. Parallel form of an information graph, its width and height. Canonical parallel form. Parallel complexity of algorithms.
14. Application efficiency dependence on the choice of data structures.

2.3.3 Online testing: knowledge check and continuing education

A properly formulated question does more than just test the level of knowledge: it allows objects and ideas to be observed and studied from another perspective, showing alternative sides of the material just learned. An extensive bank of questions was built with the authors' active involvement in the Sigma student knowledge online testing system for parallel computing [14]. Importantly, a system like this not only allows one's knowledge level to be checked for taking a test or exam. It can be successfully used for self-testing, providing students an opportunity to test how well they understand the material learned in the classroom.

All questions in the system can be divided into several categories, as we will illustrate below with a few examples ("+" marks the correct answers).

1. Questions for testing the correct understanding of the definitions.

Mark the correct statements:

- Superlinear speedup can be achieved by a large number of functional units.
- Program's scalability means that the program is suitable for parallel execution.
- A parallel program that does not possess strong scalability can possess weak scalability. +
- Efficient parallelization can be measured in terms of the number of processes running at a given moment in time.

2. Simple calculation questions.

What is the minimum time it will take to add up 512 numbers on 200 processors using pair-wise summation, if two numbers are added in 1 second and the time for a transfer data between the processors is negligible:

- 1 second
- 8 seconds
- 9 seconds
- 10 seconds +

- 11 seconds
- 200 seconds
- 384 seconds
- No correct answer.

3. Questions for analyzing the structure of algorithms or program fragments.

The multiplication of two matrices is programmed using the following fragment:

```
for(i = 0; i < n; ++i)
   for(j = 0; j < n; ++j)
      for(k = 0; k < n; ++k)
         A[i][j] = A[i][j]+B[i][k]*C[k][j];
```

What statements regarding the structure of this fragment are correct:

- The height of the canonical parallel form of the information graph for this fragment equals n. +
- The critical path length of the informational graph for this fragment equals n.
- The information graph for this fragment consists of n^2 independent computational branches. +
- This fragment cannot be executed on a computer with shared memory.
- Using any loop order in this fragment will yield the same result up to the rounding error. +
- Reordering the loops will not change the program's execution time.

4. Questions for understanding serial and parallel algorithm complexity.

A computer executed a program that multiplies two square dense matrices using a standard algorithm (without using fast multiplication methods) in 4 seconds at a performance of 32 GFlops. What were the sizes of the arrays?

- 500×500

- 1000×1000
- 2500×2500
- 4000×4000 +
- 5000×5000
- No correct answer.

2.3.4 From theory to practice

Parallel algorithms, like parallel computing in general, are an area where practice is the key. All of the theoretical ideas introduced can be easily illustrated with examples, tasks, and exercises from parallel programming practice, and by all means this should be done. Parallelism came from practice and should be explained in practical examples all the time. The experience of teaching such disciplines shows this is completely feasible even for "purely theoretical" concepts.

Various graph models of programs are considered as part of the "Structure of algorithms and programs" topic. Let's consider two types of vertices — operators (V1) or separate executions of operators (V2). For example, if a statement is executed three times in a loop, we'll get 1 operator or 3 executions of operators.

Also there are two types of edges — operational (E1) or information (E2) relationship. Vertex A is connected to vertex B with operational relationship if and only if vertex B can be executed right after vertex A. Vertex A is connected to vertex B with information relationship if and only if vertex B uses as an argument some value obtained in vertex A.

By using different methods of choosing vertices and types of relationships between them, we can derive four basic graph models: program control graph (V1+E1), program information graph (V1+E2), operational history (V2+E1) and information history (sometimes referred to as information or dependency or dataflow graph (V2+E2)). Despite the abstract nature of these concepts, they can easily be illustrated with simple examples. In particular, for one and the same example:

```
for(i = 0; i < n; ++i) {
   A[i] = A[i-1] + x;                    (1)
```

```
    B[i] = B[i] + A[i];                    (2)
}
```

all the four basic graph models are presented in Fig. 2.15.

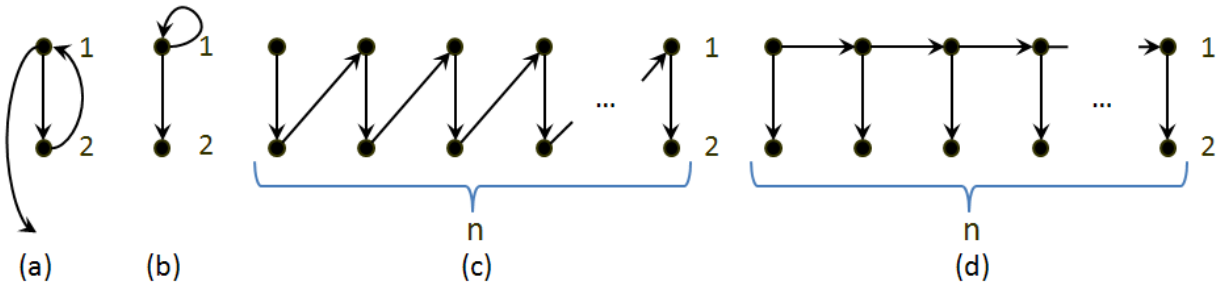


Figure (2.15) Four basic graph models: program control graph (a); program information graph (b); operational history (c); information graph (d)

To understand potential and properties of the models and to relate these ideas to program features, presentation of the material can be accompanied by a number of questions or tasks.

Task. Can the information history of a certain fragment be represented by an empty graph?

The answer is “yes”. A fragment possessing this property is shown below:

```
for(i = 0; i < n; ++i)
    A[i] = A[i] + B[i]*c;
```

Interestingly, an empty graph is far from “exotic”; on the contrary, it reflects an exceptionally important property of the program — its high degree of parallelism.

It is also important to show potential limits for the introduced concepts.

Task. Can the information history of a certain program fragment have 20 vertices and 200 edges?

Indeed, information history can be structured in different ways. But to answer this question correctly, one needs to remember its two main properties: the absence of multiple edges and its acyclic nature. This means that an information graph with the maximum number of edges will look as follows (Fig. 2.16).

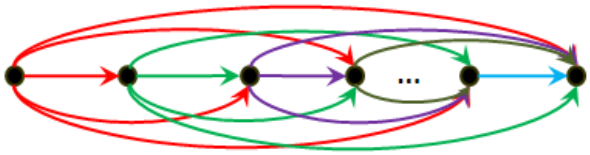


Figure (2.16) Information graph with the maximum number of edges

It follows from here that the maximum number of edges for a graph of n vertices can be calculated as $(n-1)+(n-2)+(n-3)+\cdots+2+1=n(n-1)/2$, which equals 190 for $n=20$, so the correct answer to the question above is "no".

Another type of task can help to assess the understanding of basic concepts such as the serial and parallel complexity of an algorithm.

Task. Determine the serial and parallel complexity of an algorithm implemented using the following fragment:

```
for(i = 2 ; i <= n ; ++i)
   for(j = 2 ; j <= m ; ++j)
      A[i][j] = A[i][j] * A[i][j-2];
```

The loop body will be executed $(n-1)(m-1)$ times, and each execution of the operator placed in the loop body corresponds to one multiplication operation; therefore, the total number of operations and the serial complexity equal to $(n-1)(m-1)$.

Parallel complexity is the critical path length of the fragment's information graph plus 1, where each vertex corresponds to one execution of the operator in the loop body. To determine it, we need to build an information graph, which looks as follows for this fragment (Fig. 2.17).

Obviously, parallel complexity is equal to $\lfloor m/2 \rfloor$.

The next exercise is used in the "Equivalent transformations of programs" topic.

Task. Analyze the structure and transform the following fragment for parallel execution:

```
    for(i = 1; i < n; ++i) {
1       A[i] = A[i-1]*p + q;
```

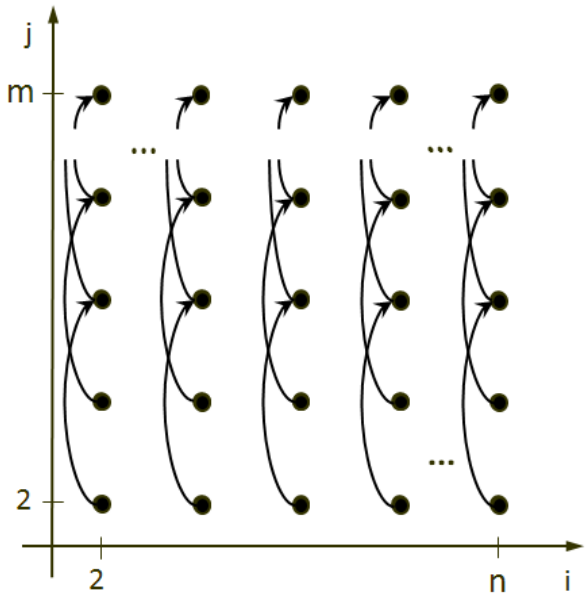


Figure (2.17) Information graph of the fragment

```
2       C[i] = (A[i] + B[i-1])*s;
3       B[i] = (A[i] - B[i])*t;
    }
```

The entire fragment's information structure is shown in the Fig. 2.18a: the fragment definitely has a substantial resource of parallelism. First let's break down the loop body to identify the serial and parallel parts within the loop body operators. The respective training course contains a statement: the necessary and sufficient condition for loop distribution is that the parts being distributed must be located in different, strongly-connected components of the information graph that represents the loop body. All three operations represent individual, strongly-connected components of the information graph, so loop distribution is possible for all three operators. The execution order for the three new loops is determined by the information structure of the information graph for the loop body: first 1, then 3, then 2, as shown on the right side of Fig. 2.18b.

Executions of operator 1 are connected with an informational dependency, so vertex 1 has a self-loop, and the corresponding loop requires serial execution. Operations 2 and 3 are not self-connected, so all iterations of these loops can be performed in parallel. The resulting fragment is

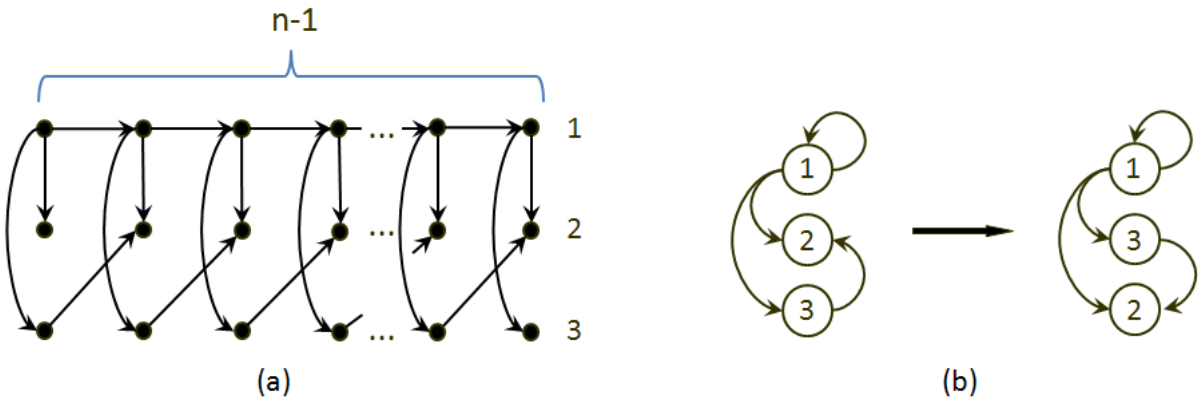


Figure (2.18) Information graph of the fragment (a) and transformation of the loop body (b)

shown below (OpenMP directives are used to declare parallel loops):

```
for(i = 1; i < n; ++i)
   A[i] = A[i-1]*p + q;
#pragma omp parallel for
for(i = 1; i < n; ++i)
   B[i] = (A[i] - B[i])*t;
#pragma omp parallel for
for(i = 1; i < n; ++i)
   C[i] = (A[i] + B[i-1])*s;
```

2.3.5 Parallel programming features

Moving students from serial programming to writing parallel programs usually does not cause major issues. Nevertheless, they need to be explicitly warned about the prospective issues that are typical for parallel algorithms and programs, to prevent them from occurring again in the future. Race condition, computational load disbalance, Amdahl's law impact — it is very helpful to review these and many other concepts.

A possible assignment for this topic would be to build a parallel implementation of a simple algorithm, while the task can focus on very different ideas. For example, build a parallel implementation of an algorithm optimized for a certain parameter, such as the utilized resource of

parallelism, execution time on a specific computer, amount of memory used, scalability, implementation efficiency, etc.

Task. A program fragment is given:

```
for(i = 1 ; i <= n ; ++i)
   for(j = 1 ; j <= n ; ++j)
      A[i][j] = A[i][j] * A[i][j] * A[i-1][j-1] ;
```

Create an implementation that uses the maximum resource of parallelism for this algorithm at any given moment.

The first thing one needs to do before building a parallel implementation is to determine the information structure of a code and identify its resource of parallelism. The information graph for this fragment will look as follows (Fig. 2.19a).

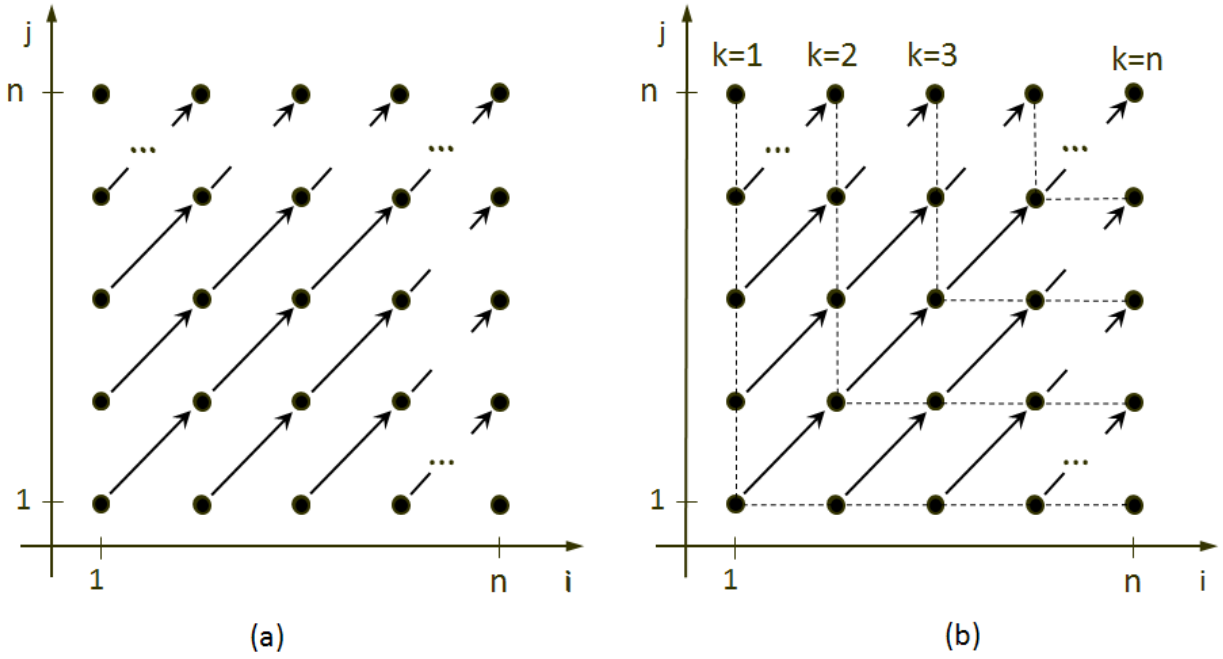


Figure (2.19) Information graph of the fragment (a) and its canonical parallel form (b)

To use the maximum possible resource of parallelism, one needs to build the canonical parallel form for this graph (it is shown in the figure 2.19b). Individual levels are shown using dashed lines.

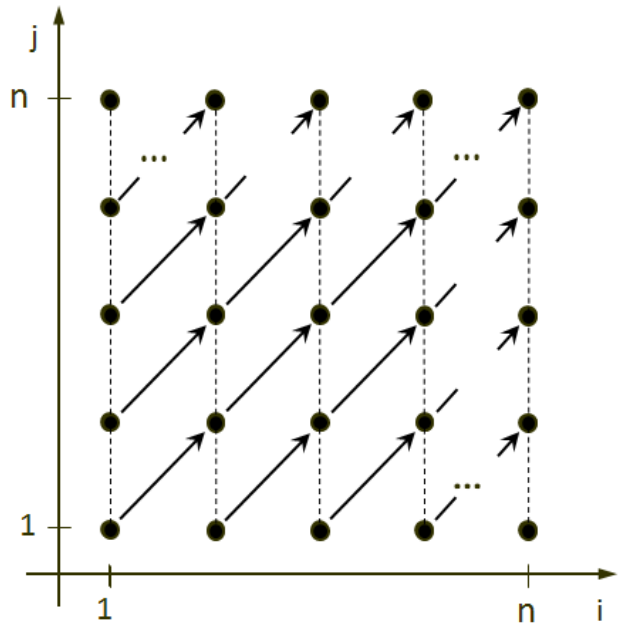
The required parallel implementation must use the entire resource of parallelism in a fragment by going through the levels in the parallel form: the total number of levels is n (this number represents the parallel complexity), and the number of vertices on each level, which can be executed in parallel, varies from $2n-1$ to 1. This approach can be implemented as follows:

```
for(k = 1 ; k <= n ; ++k){
#pragma omp parallel sections
   {
#pragma omp parallel for
      for(i = k; i <= n ; ++i)
         A[i][k] = A[i][k] * A[i][k] * A[i-1][k-1];
#pragma omp section
#pragma omp parallel for
      for(i = k+1 ; i <= n ; ++i)
         A[k][i] = A[k][i] * A[k][i] * A[k-1][i-1];
   }
}
```

This implementation also has its own nuances: the potential for using the entire resource of parallelism with this program depends on whether the OpenMP programming system supports nested parallelism.

One should recognize that this implementation is not necessarily optimal for other criteria such as execution time or code simplicity. For example, the parallel execution of the fragment can be organized using skewed parallel branches, even though this type of implementation increases parallel complexity from n to $2n-1$. At the same time, the structure of the dependencies in this example allows a very simple form of parallelism to be used for any coordinate, i or j, as in the example shown in the figure 2.20.

The parallel complexity of the implementation equals n, just like the first case, but a more convenient regular parallelism type is used with the same number of operations at every step. This version will likely be used in practice.




```
for(i = 1; i <= n; ++i)
#pragma omp parallel for
   for(j = 1; j <= n; ++j)
      A[i][j] = A[i][j] * A[i][j] * A[i-1][j-1];
```

(b)

Figure (2.20) An implementation of the fragment using coordinate parallelism

2.3.6 The efficiency of parallel applications — a matter of special attention

Creating efficient parallel applications is one of the objectives for training students in this field. It requires knowledge of parallel problem-solving methods, experience in using parallel programming technologies, and an understanding of parallel computing system architecture. A number of techniques and methodologies can be used to reinforce the material with focus on various stages of supercomputer co-design, which is the central element ensuring the efficiency of parallel applications.

Let's look at one possible version of this task which we used at the Summer Supercomputing Academy [1] held at the Lomonosov Moscow State University.

Task. Implement a parallel program on a supercomputer that multiples dense square matrices with double precision, and examine its scalability. A description of the algorithm can be found at [7]. The implementation needs to be written in a high-level programming language (C or Fortran) using MPI technology (for extra points — write a hybrid version using OpenMP inside a computing

node and MPI for communication between nodes). The task requires that none of the matrices a, b or c can be stored as a whole at any given node. Moreover, auxiliary arrays at each node can only be used to store portions of the original matrices, but not the whole matrices. The total RAM at all nodes is sufficient for storing all relevant data. The size of matrices to be multiplied in this experiment is $n = 4096$. Matrix elements of the type double (DOUBLE PRECISION) can be initialized as follows:

$$a_{ii} = b_{ii} = (n-2)/n$$

$$a_{ij} = b_{ij} = -2/n \text{ if } i \neq j$$

With these inputs, the output shall be an identity matrix, which is easy to verify. The task is to determine the correlation between program execution time (excluding initialization) and the number of processors. The number of processors p shall equal to powers of four.

First let's describe the approach to parallelization. A review of the information dependency graph for this algorithm provided in Section 1 shows that all elements of the matrix c can be computed independently from each other. Therefore, the parallel program can use a procedure to determine a single element in the resulting matrix c as the basic building block. In this case, various methods of distributing elements within the matrix c between various processes will determine the different versions of the resulting program.

Generally, an absence of information dependencies makes any distribution of the elements in the matrix c possible; however, the most natural ones are row, column and block distributions. To improve data locality, it is best to use a block distribution, where the basic block is a set parts of adjacent rows or columns (see Fig. 2.21).

In the case of MPI implementation, the distribution cannot just apply to the resulting matrix c (and related operations), but must also be determined for the original matrices a and b. This determines how much data transfer the program will need. Different versions of parallel implementation for this algorithm can be found here:

- Version 1 [8] (row distribution of matrices a and c, column distribution of matrix b);

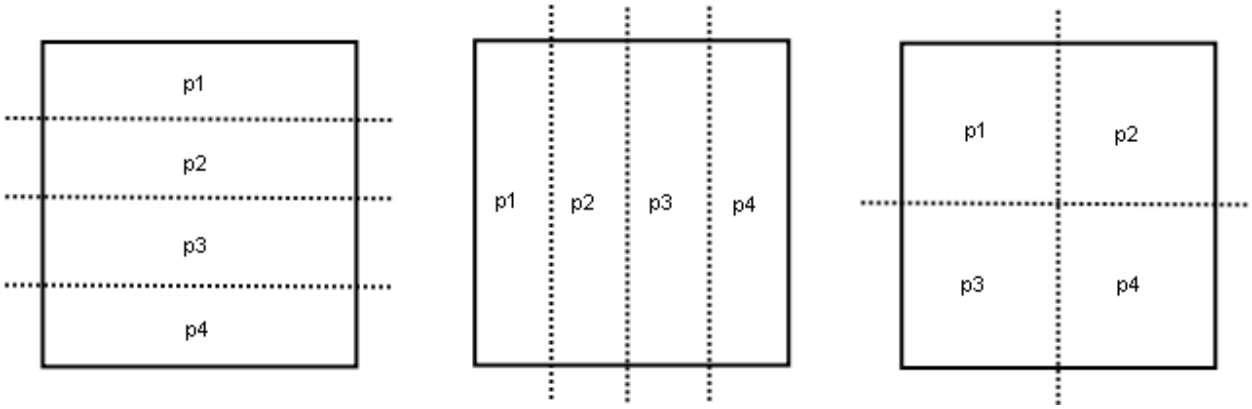


Figure (2.21) Row, column and block distributions of the matrices between 4 processes

- Version 2 [9] (block distribution of all three matrices).

In version 1, the whole row of matrix a needed to compute a certain matrix element is stored in the memory allocated to that process, while a column in matrix b may not be present in the memory for the same process, but rather is stored entirely in the memory for another process. In fact, each column in matrix b is involved in computing the entire column of matrix c, so it must be sent to all processes of the application.

In case of a block distribution into a two-dimensional process grid (version 2), each matrix dimension requires $\sqrt{p}$ processes. Matrices a, b and c are distributed between the processes in uniform blocks. For the process to be able to execute its part of operations associated with the elements of matrix c, it needs to receive data from the processes containing the rows of matrix a and columns of matrix b for the respective block.

A comparison of the two versions by the number of arithmetic operations, total volume of data transferred and the volume of data stored by each process is shown in the table 2.1. Even though both versions have the same number of arithmetic operations, version 2 requires transferring less data, but more data needs to be stored for this implementation.

A comparison of each implementation's scalability on the MSU "Lomonosov-2" supercomputer (1280 nodes using Intel Xeon E5-2697v3 processors connected with an InfiniBand FDR communication network) is shown in the table 2.2.

The examples above illustrate just a small portion of the wide variety of techniques and meth-

Table (2.1) Comparison of two versions of matrix multiplication implementations

	Number of arithmetic operations	Amount of data transferred	Volume of data stored for each process
Version 1	$2n^3$	$n^2(p-1)$	$\frac{4n^2}{p}$
Version 2	$2n^3$	$2n^2(\sqrt{p}-1)$	$\frac{3n^2+2n^2\sqrt{p}}{p}$

Table (2.2) Comparison of execution time (in seconds) for two versions of the matrix multiplication implementations on the MSU "Lomonosov-2" supercomputer

Number of processes	1	4	16	64	256	1024
Version 1	44.19	12.86	4.49	1.43	0.50	49.05
Version 2	44.66	12.76	4.44	0.84	1.05	3.05

ods that can be used in the educational process for this topic. A lot of materials for exercises can be found in the numerous algorithms described in the AlgoWiki Open encyclopedia of parallel algorithmic features [13]. At the same time, the results obtained by students while performing exercises can be used to update the encyclopedia.

Task statements can easily be modified as well. In particular, the exercises can focus on studying individual dynamic properties of programs, data locality, and various types of scalability, identifying bottlenecks in parallel implementation, building a communication profile for an application, comparing various implementations of the same algorithm, etc. Many options are possible, and they all should stimulate a creative review of the theoretical materials presented in the lecture course. There is a clear belief that obtaining practical skills in this area is as important as the theoretical study.

REFERENCES

[1] Summer Supercomputing Academy. http://academy.hpc-russia.ru/en.Cited26Jan2018

[2] Adinets, A.V., Bryzgalov, P.A., Voevodin, V.V., Zhumatii, S.A., Nikitenko, D.A., Stefanov, K.S.: Job Digest: an approach to dynamic analysis of job characteristics on supercomputers. Computational Methods and Software Development: New Computational Technologies, vol. 13, pp. 160–166 (2012)

[3] Antonov, A., Voevodin, Vad., Voevodin, Vl., Teplov, A.: A study of the dynamic characteristics of software implementation as an essential part for a universal description of algorithm properties. In 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Proceedings, pp. 359–363 (2016)

[4] Antonov, A., Voevodin, V., Dongarra, J.: Algowiki: an Open encyclopedia of parallel algorithmic features. Supercomputing Frontiers and Innovations, vol. 2, no. 1, pp. 4–18 (2015)

[5] Big Data and Extreme-scale Computing (BDEC). http://www.exascale.org/bdec.Cited26Jan2018

[6] Computer Science Curricula 2013 (CS2013). http://ai.stanford.edu/users/sahami/CS2013.Cited26Jan2018

[7] Dense matrix multiplication. http://algowiki-project.org/en/Densematrixmultiplication.Cited26Jan2018

[8] Dense matrix multiplication example, version 1. https://github.com/srcc-msu/CDER-2016/blob/master/dgemm/mpi_1d_grid.c.Cited26Jan2018

[9] Dense matrix multiplication example, version 2. https://github.com/srcc-msu/CDER-2016/blob/master/dgemm/mpi_2d_grid.c.Cited26Jan2018

[10] Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J., Barkai, D., Berthou, J., Boku, T., Braunschweig, B., et al.: The international exascale software project roadmap. International Journal of High Performance Computing Applications, vol. 25, no. 1, pp. 3–60 (2011)

[11] Supercomputing Education in Russia, Supercomputing Consortium of the Russian Universities, Tech. Rep. (2012) http://hpc.msu.ru/files/HPC-Education-in-Russia.pdf.Cited26Jan2018

[12] Future Directions in CSE Education and Research. Workshop Sponsored by the Society for Industrial and Applied Mathematics (SIAM) and the European Exascale Software Initiative (EESI-2), Tech. Rep. (2015) http://wiki.siam.org/siag-cse/images/siag-cse/f/ff/CSE-report-draft-Mar2015.pdf.Cited26Jan2018

[13] Open Encyclopedia of Parallel Algorithmic Features. http://algowiki-project.org/en.Cited26Jan2018

[14] Parallel computing collective test bank "SIGMA". https://sigma.parallel.ru/BankTest/Start/index.php?lang=en.Cited26Jan2018

[15] Prasad, S.K., Chtchelkanova, A., Dehne, F., Gouda, M., Gupta, A., Jaja, J., Kant, K., La Salle, A., LeBlanc, R., Lumsdaine, A., Padua, D., Parashar, M., Prasanna, V., Robert, Y., Rosenberg, A., Sahni, S., Shirazi, B., Sussman, A., Weems, C., and Wu, J.: NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing — Core Topics for Undergraduates, Version I. 55 pages (2012) http://www.cs.gsu.edu/~tcpp/curriculum.Cited26Jan2018

[16] Sadovnichy, V., Tikhonravov, A., Voevodin, V., Opanasenko, V.: Lomonosov: Supercomputing at Moscow State University. In: Contemporary High Performance Computing: From Petascale toward Exascale, ser. Chapman & Hall/CRC Computational Science. Boca Raton, United States: Boca Raton, United States, pp. 283–307 (2013)

[17] Scalasca. http://www.scalasca.org.Cited26Jan2018

[18] Tau Performance System. http://www.paratools.com/tau.Cited26Jan2018

[19] Vampir — Performance Optimization. https://www.vampir.eu.Cited26Jan2018

[20] Voevodin, V.: Mathematical Foundations of Parallel Computing. World Scientific Publishing Co., Series in computer science, vol. 33 (1992)

[21] Voevodin, V., Gergel, V.: Supercomputing education: the third pillar of HPC. Computational Methods and Software Development: New Computational Technologies, vol. 11, no. 2, pp. 117–122 (2010)

[22] Voevodin, V., Voevodin, Vl.: Parallel Computing. BHV-Petersburg, St. Petersburg (2002)

[23] Wirth, N.: Algorithms + Data Structures = Programs. Prentice Hall PTR (1978)