Parallel Algorithms: Theory, Practice and Education

Vl. V. Voevodin

Abstract Each new computing platform required software developers to analyze the algorithms over and over, each time having to answer the same two questions. Does the algorithm possess the necessary properties to meet the architectural requirements? How can the algorithm be converted so that the necessary properties can be easily reflected in parallel programs? *Changes in computer architecture do not change algorithms*, but this analysis had to be performed again and again when a program was ported from one generation of computers to another, largely repeating the work that had been done previously. Is it possible to do the analysis "once and for all," describing all of the key properties of an algorithm so that all of the necessary information can be gleaned from this description any time a new architecture appears? As simple as the question sounds, answering it raises a series of other non-trivial questions. Moreover, creating a complete description of an algorithm is not a challenge, it is a large series of challenges, and some of them are discussed in the paper.

1 Introduction

Parallel computing system architectures have gone through at least six generations over the past 40 years, each requiring its own algorithm properties and a special program writing style. In each case, it was important not only to find suitable features for the algorithms, but also to express them properly in the code, using special programming technologies. In fact, each new generation of computing architecture required a review of the entire software pool.

The generation of **vector pipeline computers** got off to a rapid start in the early seventies with the launch of the Cray-1 supercomputer. Machines of this class were based on pipeline processing of data vectors, supported by vector functional units and vector instructions in machine code. Full vectorization was the most efficient program implementation, which implied complete replacement of any innermost loops in the program body with vector instructions. Hence the requirements for algorithms and

Vl.V. Voevodin (✉)
Lomonosov Moscow State University, Moscow, Russia
e-mail: voevodin@parallel.ru

M.M. Resch et al. (eds.), *Sustained Simulation Performance 2016*,
DOI 10.1007/978-3-319-46735-1_1

programs: parallelism was to be expressed in the form of independent iterations of the program’s innermost loops. If that representation could be found and the innermost loops are vectored, the program would run efficiently.

During the eighties, computers came with not one, but several independent vector pipeline processors: **vector parallel computers**. The requirements for algorithm structure changed again. To support pipeline processing, inner loop parallelism was used as before. But this time, an additional parallelism resource was to be found within the algorithms that would support the independent operation of several processors. Inner loop parallelism was used to support vectorization, and outer loop parallelism to support the simultaneous operation of several CPUs.

The next generation to become commonplace during the nineties were **massively parallel distributed memory computers**, based on thousands of processors. Two actions were needed to make a program efficient. First, a substantial parallelism resource had to be identified in an algorithm to ensure the independent operation of many processors. Second, it was also important to distribute data between computing nodes to minimize data exchange during the course of program execution. This required not just another review of the algorithm pool based on the new programming technologies (with MPI becoming the de-facto standard), but also completely rewriting the software.

Shared memory computers also appeared actively. Shared memory substantially simplified the interaction between processors, making it easier to write parallel programs. Data distribution was no longer a major consideration as global address space and global shared variables eliminated many complex data handling issues. OpenMP technology appeared to reflect the new paradigm of parallel program operation. Shared memory computers required a new parallel program model, new means and methods of programming and new constructs which meant programs had to be rewritten yet again.

Computers combining the features of the two previous classes, **computing clusters with distributed memory, based on shared memory nodes**, appeared during the early 2000s. With these systems, one part of the parallelism resource inherent in an algorithm was to be kept for using a certain number of independent nodes, and the other for using several processors or cores within each node. In parallel applications, the first part was described through MPI and the second part through OpenMP. Converting an algorithm to efficiently use these features of the architecture was no trivial task, and was further complicated by the need to determine the proper data distribution for the MPI part.

About 8 years ago, **accelerators were first added** to the computer architecture—first as graphics processing units, and then as Xeon Phis. Now these devices can be found everywhere, including big clusters [2]. But what did the addition of accelerators mean for analyzing algorithm properties? It meant that a substantial parallelism resource needed to be identified in an algorithm for using many computing nodes. More parallelism needs to be present in each parallel process to utilize multiple computing cores per node. Moreover, enough parallelism needs to be left to use the accelerator features. Computers, like the need for parallelism, had become heterogeneous, which required revising the algorithm properties once more.

2 What is a Complete Description of the Algorithm Properties?

Each new computing platform required software developers to analyze the algorithms over and over, each time having to answer the same two questions. Does the algorithm possess the necessary properties to meet the architectural requirements? How can the algorithm be converted so that the necessary properties can be easily reflected in parallel programs? *Changes in computer architecture do not change algorithms*, but this analysis had to be performed again and again when a program was ported from one generation of computers to another, largely repeating the work that had been done previously.

This begs a natural question: is it possible to do the analysis "once and for all," describing all of the key properties of an algorithm so that all of the necessary information can be gleaned from this description any time a new architecture appears? As simple as the question sounds, answering it raises a series of other questions. What does it mean "to perform analysis" and what exactly needs to be studied? What kind of "key" properties need to be found in algorithms to ensure their efficient implementation in the future? What form can (or should) the analysis results take? What makes a description of algorithm properties "complete?" How does one guarantee that a description is complete and that all of the relevant information for any computer architecture is included?

The questions are indeed numerous and non-trivial. Obviously, a complete description needs to reflect many ideas: computational kernels, determinacy, information graphs, communication profiles, a mathematical description of the algorithm, performance, efficiency, computational intensity, the parallelism resource, serial complexity, parallel complexity…[3] All of these concepts, and many others, are used to describe an algorithm's properties from different perspectives, and they all are quite necessary in practice under various situations.

To immediately introduce some order to these diverse concepts, one can begin by breaking up an algorithm's description into two parts. The first part is dedicated to the algorithm's theoretical properties, and the second part describes its particular implementation features. This division allows the machine-independent properties of algorithms to be separated from the numerous issues arising in practice. Both parts of the description are equally important: the first one describes the algorithm's theoretical potential, and the second one demonstrates the practical use of that potential. The first part of the description explains the mathematical formulation of the algorithm, its computational kernel, input and output data, information structure, parallelism resources and properties, determinacy and computational balance of the algorithm, etc. The second part contains information on an algorithm's implementation: locality, performance, efficiency, scalability, communication profile, implementation features on various architectures, and so on.

3 Why Is It Hard to Describe Algorithms?

Many of the ideas described above are very well known. However, as you start describing the properties of real algorithms, you realize that creating a complete description of an algorithm is not a challenge, it is a large series of challenges! Unexpected problems arise at each step, and a seemingly simple action becomes a stumbling block. Let’s look at the information structure of an algorithm mentioned above. It is an exceptionally useful term that contains a lot of information about the algorithm. An information graph is a convenient representation of an algorithm’s information structure. In many cases, looking at the information graph is enough to understand its parallel implementation strategy. Figure 1a, b show the information structure for typical computational kernels in many algorithms, Fig. 1c shows the information structure of a Cholesky decomposition algorithm.

An information graph can be simple for many examples. However, in general, the task of presenting an information graph is not a trivial exercise. To begin with, a graph can potentially be infinite, as the number of vertices and arcs is determined by the values of external input variables which can be very large. In this situation it helps to look at likenesses: graphs for different values of external variables look very “similar” to one another, so it is almost always enough to present one small graph, stating that the 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 arcs in an information graph relies

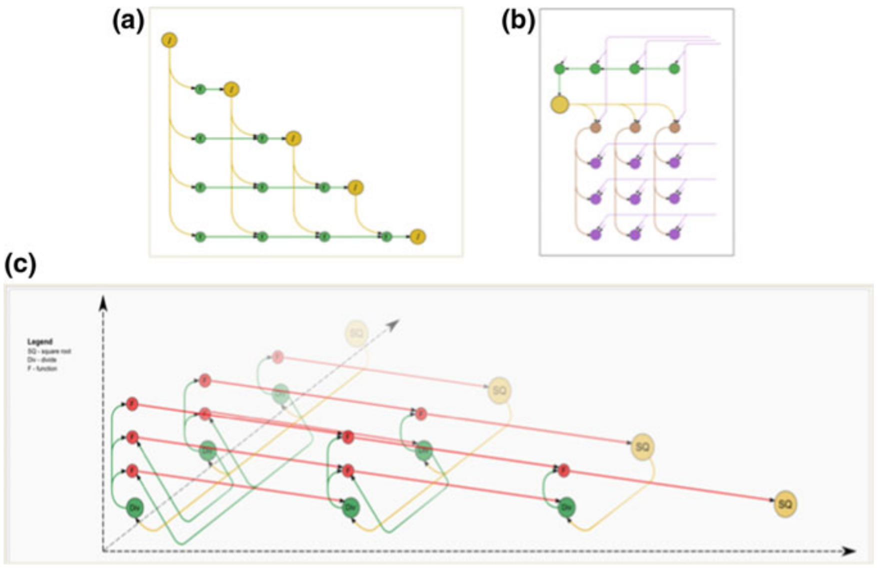

Fig. 1 Information structure of various algorithms

on the nested loops in an algorithm's implementation. If nested loops do not go deeper than three levels (as in a classical matrix multiplication algorithm), the graph can be placed in the traditional three-dimensional space. More complex looping constructs with 4 or more nesting levels require special methods for presenting and displaying the graph. But even if the number of dimensions does not exceed three, how does one make the graphical presentation informative? Figure 2a shows a graph in its entirety, which is barely comprehensible. Various projections of the same graph are shown in Figs. 2b, d, which can help assess an algorithm's parallelism potential, but these aren't always helpful…

Related questions also arise: how does one visualize the parallelism potential and illustrate parallel implementation methods for the algorithm? Sometimes a canonical parallel layer form [3] comes in handy, which reflects both the algorithm's parallel complexity and the fastest method for its parallel implementation (within the infinite parallelism concept), but it is very difficult to build and not always feasible. Figure 3 shows the sequential execution of the fragment in Fig. 2a in five steps. Red indicates vertices within the current level that can be executed in parallel on the current step. Green indicates vertices executed in previous steps, and white indicates vertices that can only be executed in subsequent steps. By visualizing the step-by-step sequential movement of the red vertices, one can evaluate the parallelism available at each step. How does one find, analyze, describe and display the canonical parallel layer form? The question remains open for arbitrary programs.

The issues of data locality and computation locality are of paramount importance in describing an algorithm's properties and its implementations. Locality is

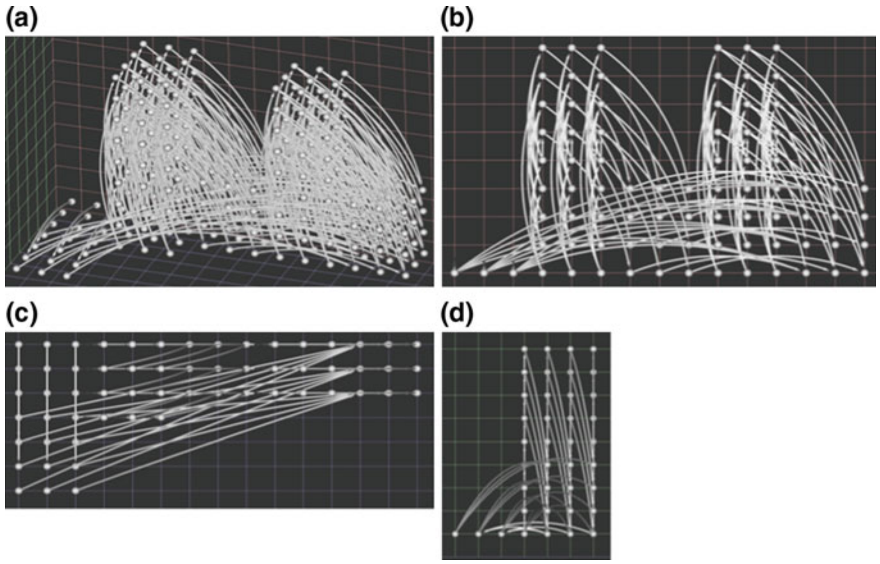

Fig. 2 Methods of displaying an algorithm's information structure

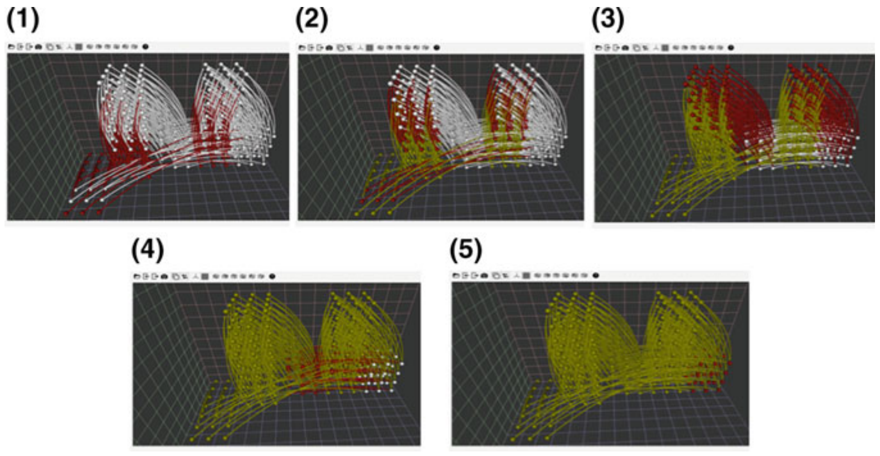

Fig. 3 Sequence of steps in the parallel execution of an algorithm based on a canonical parallel layer form

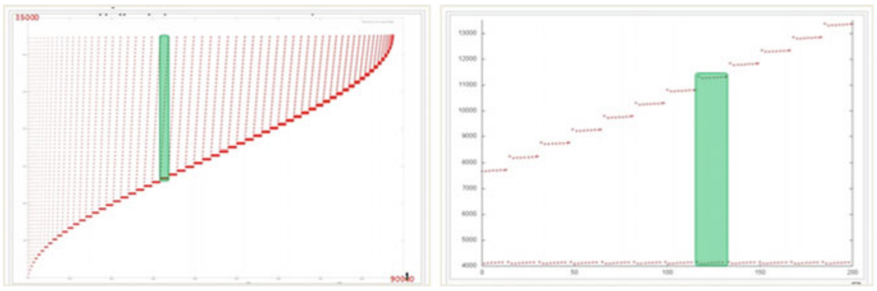

Fig. 4 A description of data locality in programs using memory access profiles

what determines program execution efficiency on modern computing platforms. To get the complete picture of an algorithm's particular implementation features, it is important to analyze both temporal and spatial locality, noting positive and negative factors related to locality, and the conditions and situations by which they are caused. However, even a quick look makes it obvious that there are many more questions than answers in this area. What methodology can be used to evaluate the temporal and spatial data locality in the programs? How can one compare temporal and spatial data locality between different programs? Figure 4 shows the memory access profiles for two programs, indicating the memory address after each memory access operation. Which program has better temporal and/or spatial data locality? In some cases, memory usage templates help: they are simple and their characteristics are predetermined, but once again, the issue of carefully studying locality properties in an arbitrary program generally remains open.

Another interesting question is related to how data locality is related to the algorithm structure. In other words, can we predict data locality in a given program by using just the information about its algorithm? On the one hand, there are no data structures in algorithms—they only appear in programs; so talking about locality for algorithms is not exactly right. On the other hand, it is the algorithm that determines the structure and properties of a program to be coded, including its locality. Many have probably heard the expression "the algorithm's locality" or "this algorithm has better locality than the other." How appropriate are these statements, given that algorithms do not contain data structures?

Determinacy is an important practical aspect of algorithms and programs, but how can one describe all of the potential sources which violate this property? A serious cause of indeterminacy in parallel programs is related to changes in the order of executing associative operations. A typical example is the use of global operations in Message Passing Interface (MPI) by a number of parallel processes, e.g., when summing the elements of a distributed array. The MPI runtime system chooses the order of execution on its own, assuming compliance with the associative law, which results in various round-off errors and ultimately in different results when executing the same application. This is a serious issue often encountered in massively parallel computing systems that causes results of parallel program execution to not be reproducible. If the analysis of an algorithm's structure shows that the resulting parallel application cannot work without global operations, this property must be included in the algorithm description. To analyze this problem properly, a communication profile should be built for the parallel program, pointing out the structure and interaction method between parallel processes. A clear definition of the communication profile hasn't been produced to date, so it is premature to consider in-depth analysis in this area.

Indeed, there are many open questions, and the list can go on. The main question that still remains unanswered is "What does it mean *to create a complete description of an algorithm?*" What must be included in this description, so that we can glean all of the necessary information from it every time a new computing platform appears? The task seems simple at first sight: an algorithm is just a sequence of mathematical formulas, often short and simple, which should easily be analyzed. But at the same time, no one can guarantee the completeness of such a description.

The properties of the algorithms and programs discussed in this work became the foundation for the AlgoWiki project [1]. The project's main goal is to provide a description for fundamental algorithm properties which will enable a more comprehensive understanding of their theoretical potential and their implementation features in various classes of parallel computing systems. The project is expected to result in the development of an open online encyclopedia based on wiki technologies which will be open to contributions by the entire academic and educational community. The first version of the encyclopedia is available at http://AlgoWiki-Project.org/en, where users can describe both their own pedagogical experience and their knowledge of specific parallel algorithms.

4 Conclusion

All of the issues discussed in this work are highly important for training future specialists [4–6]. Right from the beginning of the education process, focus should be placed on algorithm structure since it determines both the implementation quality and the potential for efficiently executing programs in a parallel environment. The algorithm structure and its close relationship to parallel computing system architecture are central ideas in parallel computing, which are included in many courses for Bachelor's and Master's degree programs at the Faculty of Computational Mathematics and Cybernetics at Lomonosov Moscow State University, as well as in the lectures and practical courses offered by the annual MSU Summer Supercomputing Academy [7]. We are also trying to expand this concept to the Supercomputing Consortium of Russian Universities [8] in order to develop a comprehensive supercomputer education system, rather than offering occasional training aimed at rectifying the situation.

Acknowledgements This project is being conducted at Moscow State University with financial support from the Russian Science Foundation, Agreement No 14-11-00190.

References

1. Antonov, A., Voevodin, V., Dongarra, J.: AlgoWiki: an open encyclopedia of parallel algorithmic features. Supercomput. Front. Innov. **2**(1), 4–18 (2015)
2. Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.C., Barkai, D., Berthou, J.Y., Boku, T., Braunschweig, B., et al.: The international exascale software project roadmap. Int. J. High Perform. Comput. Appl. **25**(1), 3–60 (2011)
3. Voevodin, V.V., Voevodin. Vl.V.: Parallel Computing. BHV-Petersburg, St. Petersburg (2002). (in russian)
4. Computing Curricula Computer Science. http://ai.stanford.edu/users/sahami/CS2013 (2013)
5. 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), http://wiki.siam.org/siag-cse/images/siag-cse/f/ff/CSE-report-draft-Mar2015.pdf (2015)
6. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing. http://www.cs.gsu.edu/~tcpp/curriculum/
7. Summer Supercomputing Academy. http://academy.hpc-russia.ru/
8. Supercomputing Education in Russia, Supercomputing Consortium of the Russian Universities. http://hpc.msu.ru/files/HPC-Education-in-Russia.pdf (2012)