Parallel Programs Execution Optimization Using Behavior Control in LuNA system

Victor Malyshkin[1,2,3], Darkhan Akhmed-Zaki[4[0000-0001-8100-8263]] and Vladislav Perepelkin[1,2[0000-0002-6998-4525]]

[1] Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Novosibirsk, Russia

[2] Novosibirsk State University, Novosibirsk, Russia

[3] Novosibirsk State Technical University, Novosibirsk, Russia

[4]Astana IT University, Nur-Sultan, Kazakhstan

perepelkin@ssd.sscc.ru

Abstract. In the paper the problem of efficient parallel execution of numerical algorithms for supercomputers in the LuNA system is concerned. With LuNA an application algorithm is represented in a hardware-independent high-level form. This allows implementing the algorithm by automatic construction of various parallel programs, which possess different non-functional properties, such as execution time, memory consumption, network workload, etc. In the LuNA system the efficiency problem of automatically constructed parallel programs is dealt with through the behavior concept. The presented approach allows controlling parallel program behavior without low-level programming of the desired behavior.

Keywords: Automatic Parallel Programs Construction, Fragmented Programming Technology, LuNA System, Parallel Program Behavior.

1 Introduction

During the last decades supercomputers are being constantly improved, their computational potential is being increased. However, realization of the potential for large-scale numerical simulations becomes more complex and troublesome due to the hardware complexity increase. The number of computing nodes and cores is increasing. Memory, network and computing nodes configuration heterogeneity is also increased. Computing nodes fault probability is increased with nodes count increase. GPUs and other co-processors usage becomes essential in achieving the highest performance. In order to achieve satisfactory efficiency of application software, developers have to take into account all these details. In many cases some additional functionality, such as checkpointing or dynamic load balancing, is required. Development of such software is hard and complicated, especially for most supercomputer users, who are not experts in system parallel programming.

Especially this problem affects users, who develop new numerical models and algorithms, and therefore they are unable to use ANSYS Fluent [1], NAMD [2] or other highly-efficient software tools, optimized by skillful system programmers. Strict performance and memory constraints also make unusable most mathematical software, oriented at non-programmers. Examples of such software are MathWorks MATLAB [3], GNU OCTAVE [4] or Wolfram Mathematica [5].

Many programming languages, systems and tools exist [6-11] to reduce the complexity of parallel programs development. With such means an application programmer describes an application algorithm in a high-level form. The description is then used to automatically construct a parallel program, which implements the algorithm. The program constructed is often optimized for efficiency and possesses dynamic properties, such as fault tolerance, dynamic load balancing, etc. If necessary, the program can be reconstructed automatically for different execution conditions (hardware configuration, data size, non-functional requirements, etc.). Use of such languages, programming systems, and tools simplifies parallel programs development and maintenance, hides the complexity of low-level system programming tasks. Automatically constructed programs can sometimes outperform manually developed programs. At least, the automation reduces programs' development and modification laboriousness.

In Charm++ [6] computations are represented as a set of distributed communicating objects called chares. The run-time environment is capable of serializing and redistributing chares, scheduling their execution and performing other execution management tasks in order to optimize program execution efficiency. A user is allowed to tune some settings of the execution, including choice of a dynamic load balancer. Charm++ achieves high efficiency while freeing the user from a number of complex tasks of low-level parallel programming. In PaRSEC [7] the subject domain is limited to a dense linear algebra algorithms class (and some other similar algorithms). In particular, iterations with dynamic conditions are not supported. This and other constraints are used to make particular systems algorithms and heuristics effective. This, in turn, allows achieving high performance within the subject domain. Legion [8] system follows a powerful approach to define computations separately from execution and resources management. With this approach the user is responsible for programming resources distribution, computations scheduling and other execution management tasks. Legion provides means to do it without the risk of bringing errors into code. LuNA system [9] follows a similar approach, but the management code can also be generated automatically. Many other systems exist and evolve to study various computational models, system algorithms and heuristics and develop better facilities of parallel programs construction automation [10,11].

To provide a higher level of programming for users, programming systems have to deal with algorithmically hard problems, such as such as computations scheduling, resources distribution, etc. This forces the systems to employ various heuristic approaches to provide satisfactory performance for a limited class of applications and hardware. Big effort is being put into development of such systems, and much more work has to be done to improve the quality of provided automation.

This paper discusses the approach, which is employed by the LuNA system to achieve satisfactory performance of constructed parallel programs. The LuNA system is a part of an academic project of the Institute of Computational Mathematics and Mathematical Geophysics of the Siberian Branch of Russian Academy of Sciences. The project is devoted to automation of large-scale numerical simulation programs construction. With LuNA a user is able to describe an application algorithm in a domain-specific language. The description is then used by to system to automatically construct a parallel program, which implements the algorithm. LuNA follows the active knowledge paradigm [12] and is based on the theory of parallel programs synthesis on the basis of computational models [13].

The next sections present a brief introduction to the LuNA system and the behavior concept, which is employed to concern the efficiency problem. Some performance tests and the conclusion section end the paper.

2 The LuNA System

With LuNA an application programmer essentially describes an application algorithm and its decomposition in a form called fragmented algorithm (FA). The algorithm is described in a hardware-

independent form as a set of aggregated operations called computational fragments (CFs), and a set of aggregated variables called data fragments (DFs). Each CF is bound to a finite number of input and output DFs. This forms a potentially infinite directed acyclic bipartite graph of CFs and DFs. DF is an immutable (single assignment) piece of data, formed as the result of the algorithm's data decomposition. CF is an application of a pure function (with no side-effects) to values of CF's input DFs to compute values of CF's output DFs. Initially a subset of DFs get their values set (the DFs are called FA input). Then the FA is executed in a data-flow manner, i.e. the CFs are executed asynchronously as their input DFs are getting computed. The process stops when no CFs can be executed. Note, that CFs can be executed in any order (including in parallel), which does not violate data dependencies. Since DFs are immutable and CFs are pure functions the order of CFs execution has no influence on DFs' values. Such representation is based on the concept of computational model [13], i.e. FA is a particular form of computational model, where exactly one computational plan is deductible.

Parallel execution of a FA on a multicomputer (distributed memory computer) is based on the following idea. All CFs should be mapped to the set of computing nodes of the multicomputer. Also, each CF should have a sequential module (subroutine) assigned. The modules are used to implement CFs. To execute a CF all its input DFs (which must be computed first) are transferred to the node, the CF is mapped to. Then the CF is executed locally by applying the module to the CF's inputs, to produce the values of its output DFs. Then output DFs are transferred to other nodes to be processed by other CFs, and so on. Since all data dependencies are explicit, CFs have no side effects and DFs are immutable, FA parallel execution can be performed automatically. The programming system can automatically handle such tasks as resources management, processes and threads synchronization, network communication, scheduling, garbage collection and others. Even co-processor offloading can be automated if a compatible module for executing a CF on the co-processor is provided. Checkpointing, dynamic load balancing and other dynamic properties can be automatically provided by saving and loading DFs to storage or send them via the network. Many optimization techniques, such as DFs replication, CFs execution reordering, speculative execution, etc. can be applied automatically without the risk of disrupting the computations, described as a FA.

However, these extensive abilities of FA execution do not yet guarantee satisfactory efficiency of FA execution. A system not only can, but also must dynamically distribute CFs and DFs to nodes, choose CFs' execution order, collect garbage, and solve other problems. The solutions affect non-functional properties of the FA execution. Many of the problems are algorithmically hard, so various heuristics and particular approaches are to be used. This way satisfactory performance can be achieved for a limited classes of applications and hardware. This is common for all high-level programming systems.

There are two basic approaches to FA execution. The first approach is to dynamically interpret FA by a run-time system. With this approach a distributed virtual machine (a run-time system) is running on a multicomputer. It takes the FA and input data as input. The run-time system manages distributed objects of two kinds - CFs and DFs. The system transfers CFs and DFs from one computing node to another in order to gather a CF and all its input DFs on a single node and execute the CF (invoke the module which implements it) to produce new DFs. The run-time system distributes and dynamically redistributes CFs and DFs to computing nodes to equalize workload, chooses order of CFs execution (according to data-dependencies) and performs various optimizations. The run-time system is able to provide checkpointing and other useful mechanisms if necessary.

The other approach is compilation. A FA is statically translated into a conventional parallel program. The program is generated according to the control logic, imposed by FA description and consists mostly of DFs network transfers and invocation of modules, which implement CFs.

The run-time system approach is advantageous in sense that the run-time system can dynamically react to peculiarities of the execution process. The peculiarities are conditioned by input data peculiarities, hardware configuration and other dynamic factors. Advantage of the compiler approach is the absence of run-time system overhead, which can be significant. The two approaches can be combined in various combinations. For example, a statically generated program may contain dynamic workload balancing support, as well as a run-time system may execute a statically modified FA. For example, multiple CFs can be "fused" into a larger CFs to reduce the overhead. In practice, particular configuration of the generated program and its run-time support system should vary, depending on many factors, such as application peculiarities, data peculiarities, hardware configuration, particular non-functional requirements and so on. Thus, a FA can be implemented in a wide variety of ways, which differ in resources distribution, order of CFs execution, garbage collection strategies, dynamic properties support, etc. All these generated programs can possess different non-functional properties, different efficiency: memory consumption, network workload, computational time and other characteristics may vary significantly. The main problem for the LuNA system is to provide desired non-functional properties and satisfactory efficiency.

3. The Behavior Concept

One of the key concepts employed in LuNA to concern the efficiency problem is the behavior concept. FA behavior is the set of all possible valid executions of the FA. While the values of DFs are the same among all the executions, other characteristics differ: distribution and redistribution of DFs and CFs to computing nodes, order of CFs execution, garbage collection strategy, network routing, etc. In particular, the non-functional properties of interest (execution time, memory consumption, network load, etc.) differ from one behavior element to another. So the efficiency problem can be formulated as a problem of behavior reduction to a subset, which contains only elements with satisfactory non-functional properties.

The work of the LuNA system at compile time or run time consists of making and implementing decisions. For instance, the compiler may decide to execute two CFs *a* and *b* on the same computing node, because they process the same input DFs. Later the run-time system decides, that *a* and *b* will be executed on the computing node *N*. Both decisions are constraints, which reduce the FA behavior by excluding all elements which do not fit into the constraints. The same can be said about all other decisions the LuNA system makes: they reduce the behavior. Note, that making good decisions is hard, while implementing the decisions is relatively straightforward.

The behavior concept is explicit in LuNA. A FA defines the widest behavior. Then the compiler adds supplementary information, which contains statically made decisions on FA execution. This reduces the behavior to its subset. Then the run-time system takes both FA and the static decisions as input and makes dynamic decisions, reducing the behavior further. In the end of FA execution the reduced behavior contains a single element, which is exactly the execution occurred. This element can be investigated using profiling and other techniques.

The efficiency problem, which is formulated as the behavior reduction problem, is hard to solve automatically. It is hard to predict which elements of the behavior possess satisfactory non-functional properties. That is why human help is of great importance. The behavior concept allows a user to supply additional information called recommendations. Recommendations are basically means to partially reduce behavior. Using recommendations the user can express an idea of how FA should be executed without the need to hard-code this behavior into the program. E.g. the user can recommend to distribute some DFs in a particular way, suggest garbage collection strategy, partially define order of CFs execution, etc. Such recommendations help LuNA to achieve better efficiency. It is important that the user can not disrupt the FA execution in the sense of DFs' values computed, since behavior only affects non-functional properties. This, in particular, allows to safely involve an external expert to optimize the FA execution, which is not possible with traditional approaches. Also user recommendations can be discarded by LuNA if it is able to find a better behavior. Thus both human and machine can contribute to efficiency improvement,.

The behavior can be analyzed in details in terms of fragments. For example, there is no problem to collect profiling information on fragments' distribution to nodes, CFs' execution order and time, DFs' size and migration routes, etc. All this information can be visualized for human analysis or processed automatically for compiler and run-time system profile-based optimizations.

Such an orthogonality of FA and recommendations is common for various programming systems [6–9], since it is the basis, which allows a system to control and optimize execution. In some systems, such as Charm++, the system distributes the objects using system algorithms. In other systems, such as PaRSEC, the user specifies the distribution without programming it, and the system implements it. In systems, such as Legion the user needs to program the distribution using system API.

4 Performance Evaluation

This section presents performance tests for a model 3D heat equation solution in a unit cube. This application was studied in our previous papers [14, 15], where more details on the application can be found. The application data consists of a 3D mesh, decomposed in three dimensions into subdomains. The computations are performed iteratively, where each step is solved with pipelined Thomas algorithm [16].

The testing was conducted on the MVS-10P supercomputer of the Joint Supercomputer Centre of Russian Academy of Sciences [17]. It comprises 2×Xeon E5-2690 CPU-based computing nodes with 64 GB RAM each. The following parameters, representative for such applications, were chosen. Mesh size: from 100^3 to 1000^3 with step 100 (in every dimension), number of cores: 6^3 (216).

In the test three implementations of the same algorithm are presented. The first one (L1) is an initial LuNA implementation of the algorithm. Then behavioral profiling analysis was employed to identify inefficient resources usage and recommendations were added to optimize efficiency (L2). Also a reference hand-coded MPI (Message Passing Interface) implementation of the same algorithm was tested (MPI). The results are shown in Fig. 1.

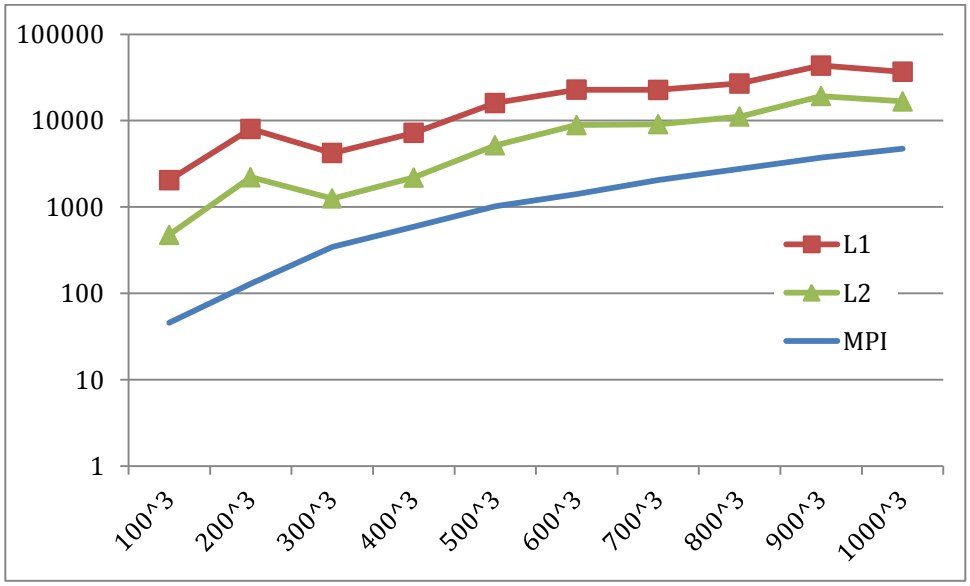

Fig. 1. Program execution time (in seconds). L1, L2 and MPI are LuNA, LuNA optimized and reference MPI implementations correspondingly. The number denotes the number of cores. The X axis is the mesh size.

Fig. 1 shows that behavioral analysis accompanied with recommendation-based behavior control is an effective approach to optimizing FA execution, which does not involve low-level programming. It also can be seen that MPI implementation demonstrates better efficiency, which is caused by two main reasons. Firstly, the behavior can be investigated and optimized further to obtain better efficiency. Secondly, compiler and run-time system algorithms and their software implementations can be optimized to reduce run-time overhead. The reference MPI implementation is about 10 times faster, although such a slowdown of automatically constructed programs may often be tolerable, because development of FA requires less skill and effort from the programmer than MPI-program development. Also, LuNA is being constantly improved, which causes more efficient execution of existing FAs' without any need to change them.

5 Conclusion

An approach to achieve efficient parallel execution of numerical simulation algorithms, defined in a high level language, is considered, as well as its implementation in the LuNA system. The behavior concept is concerned as a means of efficiency optimization. Performance tests were conducted to illustrate the approach on a practical example. In the future both software optimization and intelligent algorithms and tools of behavior optimization should be conducted to achieve better performance.

This research has was funded by the budget project of the ICMMG SB RAS No 0315-2019-0007 and by the Science Committee of the Ministry of Education and Science of the Republic of Kazakhstan AP05134651 "Development active knowledge control system for automation of construction of high performance parallel programs of unstructured data processing and numerical modeling in filtration problems."

References

1. ANSYS Fluent Web Page, https://www.ansys.com/products/fluids/ansys-fluent, accessed: 2019/04/01.

2. Phillips, J., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R., Kale, L. Schulten, K.: Scalable molecular dynamics with NAMD. Journal of Computational Chemistry, 26:1781-1802 (2005).

3. MathWorks MATLAB official web-site, https://www.mathworks.com/products/matlab.html, last accessed: 2019/04/01.

4. GNU Octave Web Site, https://www.gnu.org/software/octave/, last accessed: 2019/04/01.

5. WOLFRAM MATHEMATICA Web Site, http://www.wolfram.com/mathematica/, last accessed: 2019/04/01.

6. Robson, M., Buch, R., Kale, L.: Runtime Coordinated Heterogeneous Tasks in Charm++. In: Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware (2016).

7. Wu W., Bouteiller A., Bosilca G., Faverge M., Dongarra J.: Hierarchical DAG Scheduling for Hybrid Distributed Systems. In: 29th IEEE International Parallel & Distributed Processing Symposium (2014).

8. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing Locality and Independence with Logical Regions. In: the International Conference on Supercomputing (SC 2012) (2012).

9. Malyshkin, V., Perepelkin, V.: LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem. In: Parallel Computing Technologies. LNCS 6873, pp. 53–61 (2011).

10. Sterling, T., Anderson, M., Brodowicz, M.: A Survey: Runtime Software Systems for High Performance Computing. Supercomputing Frontiers and Innovations: an International Journal, 4(1), pp. 48–68. (2017). DOI: 10.14529/jsfi170103.

11. Thoman, P., Dichev, K., Heller, T. et al.: A taxonomy of task-based parallel programming technologies for high-performance computing. The Journal of Supercomputing, 74(4), pp. 1422–1434. (2018). DOI: 10.1007/s11227-018-2238-4.

12. Victor E. Malyshkin. Literacy for Oncoming Centuries // In the Proceedings of the 13th International Conference on Intelligent Software Methodologies, Tools, and Techniques (SoMeT), Book Series: Frontiers in Artificial Intelligence and Applications, Vol. 246, IOS Press, USA, 2014, pp. 899-905. DOI: 10.3233/978-1-61499-434-3-899

13. Valkovsky, V., Malyshkin, V.: Synthesis of parallel programs and systems on the basis of computational models. Nauka, Novosibirak (1988).

14. Akhmed-Zaki, D., Lebedev, D., Perepelkin, V. J Supercomput (2018). https://doi.org/10.1007/s11227-018-2710-1

15. Akhmed-Zaki, D., Lebedev, D., Malyshkin, V., Perepelkin, V. Automated construction of high performance distributed programs in LuNA system // 15th International Conference on Parallel Computing Technologies, PaCT 2019; Almaty; Kazakhstan. LNCS 11657. Springer, 2019. pp. 3-9. DOI: 10.1007/978-3-030-25636-4_1.

16. Sapronov, I., Bykov, A.: Parallel pipelined algorithm. Atom 2009, no 44, pp 24–25 (2009) (in Russian)

17. Joint Supercomputing Centre of Russian Academy of Sciences Official Site, http://www.jscc.ru/, last accessed: 2019/04/01