Automated Construction of High Performance Distributed Programs in LuNA System

Darkhan Akhmed-Zaki[1], Danil Lebedev[1][0000-0002-5186-6483], Victor Malyshkin[2,3,4] and Vladislav Perepelkin[2,3][0000-0002-6998-4525]

[1] Al-Farabi Kazakh National University, Almaty, Kazakhstan
[2] Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Novosibirsk, Russia
[3] Novosibirsk State University, Novosibirsk, Russia
[4] Novosibirsk State Technical University, Novosibirsk, Russia
perepelkin@ssd.sscc.ru

Abstract. The paper concerns the problem of efficient distributed execution of fragmented programs in LuNA system, which is a automated parallel programs construction system. In LuNA an application algorithm is represented with a high-level programming language, which makes the representation portable, but also causes the complex problem of automatic construction of an efficient distributed program, which implements the algorithm on given hardware and data. The concept of adding supplementary information (recommendations) is employed to direct the process of program construction based on user knowledge. With this approach the user does not have to program complex distributed logic, while the system makes advantage of the user knowledge to optimize program and its execution. Implementation of this concept within LuNA system is concerned. In particular, a conventional compiler is employed to optimize the generated code. Some performance tests are conducted to compare efficiency of the approach with both previous LuNA release and reference hand-coded MPI implementation performance.

Keywords: Automated Parallel Programs Construction, Fragmented Programming Technology, LuNA System.

1 Introduction

Considerable constant growth of supercomputers' capabilities during last decades is accompanied with the increase of complexity of high performance computing hardware usage. This, in turn, makes implementation of large-scale numerical models harder for users of supercomputers. Efficient utilization of modern supercomputers' resources requires an application to be scalable and tunable to hardware configuration. In some cases dynamic load balancing, co-processors support (GPU, FPGA, etc.), fault tolerance and other properties are required. Implementation of such properties is not easy and requires specific knowledge and skills, different from what implementation of the "numerical" part of the program requires.

Especially this problem affects users, who develop new numerical models and algorithms, and therefore they are unable to use ANSYS Fluent [1], NAMD [2] and other highly-efficient software tools, optimized by skillful programmers. Strict performance and memory constraints also make unusable most non-programmer friendly mathematical software, such as MathWorks MATLAB [3], GNU OCTAVE [4] or Wolfram Mathematica [5]. The only option to reduce complexity of efficient parallel program development is to employ programming systems [6–11], which automate many low-level error-prone routine jobs and provide higher level means, suitable for various particular cases.

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. The user is allowed to tune some settings of the execution, including choice of dynamic load balancer. Charm++ achieves high efficiency while freeing the user from a number of complex tasks of parallel programming. In PaRSEC [7] the application 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, which, in turn, allows to achieve high performance within the application domain. Legion [8] system follows a powerful approach to separately define computations and their execution management as orthogonal parts of the application. With this approach the user is responsible for programming resources distribution, computations scheduling and other execution management tasks, but the means Legion provides allow doing it without the risk of bringing errors into code. LuNA system [9] follows the similar approach, but instead of obliging the user to do the management the system allows automated construction of the management code. 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].

It can be stated, that big effort is put into development of such systems, although much more work has to be done in order to widen their application domains and improve quality of the automation performed.

This paper discusses the approach employed by LuNA system to achieve satisfactory performance of constructed parallel programs. LuNA is a system of automated construction of parallel programs, which implement large-scale numerical models for supercomputers. The system is being developed in the Institute of Computational Mathematics and Mathematical Geophysics, SB RAS.

The next sections present the fragmented programming technology approach upon which LuNA system is based, the implementation of the approach in LuNA system and some performance tests. The conclusion and future works section ends the paper.

2 The Fragmented Programming Technology Approach

In the fragmented programming technology an application algorithm is represented in a hardware-independent form called fragmented algorithm. Fragmented algorithm (FA) is basically a declarative specification, that defines two potentially infinite sets — a set of computational fragments (CF) and a set of data fragments (DF), where each CF is a side-effect free sequential subroutine invocation and each DF being an immutable piece of data. For each CF two finite subsets of DFs are defined to be input and output DFs correspondingly. The CF's subroutine computes values of output DFs provided values of input DFs are available in local memory. FA as an enumeration representation employs a number of operators, which describe DFs and CFs. The representation is based on the definition of computational model [12], i.e FA is a particular form of computational model, in which exactly one algorithm is deductible.

Fragmented program (FP) is an FA with supplementary information called recommendations. While FA defines computations functionally (i.e. how DFs are computed from other DFs), recommendations affect non-functional properties of the computations, such as computational time, memory usage, network traffic, etc. For example, a recommendation may force two DFs to share the same memory buffer within different time spans in order to reduce memory usage, or a recommendation may define data allocation strategy for a distributed array of DFs, etc.

FA and recommendations are orthogonal in sense that recommendation do not affect the values computed, but only affect how FA entities (DFs and CFs) are mapped into limited resources of a multicomputer in time. Different recommendations cause execution of the same FA to be optimized for different hardware configuration and/or optimization criteria (memory, time, network, etc.). There are two different kinds of recommendations. The first one is informational recommendation, which formulates properties of FA, which are hard to obtain automatically, for example estimated computational complexity of different CFs or the structure of DFs. The second kind of recommendations is prescriptive recommendation, which directs the execution in some way, for example mapping of DFs to computing nodes or order of CFs execution. Neither kind of recommendations is mandatory and even if recommendations are supplied, they can be partially or completely ignored by the system.

Such an orthogonality is common for various programming systems [6–9], since it is the basis, which allows a system to control execution. Let's illustrate some differences in systems' approaches on the example of objects (fragments, jobs, etc.) distribution. 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.

In LuNA a hybrid approach is employed. If no recommendations are supplied, the system will decide on distribution using system algorithms. If informational recommendations are supplied, a (probably) better distribution will be constructed based on this additional knowledge. If prescriptive recommendations are given, then they will be followed by the system. The prescriptive recommendations are least portable, they are useful until the system is able to automatically construct satisfactory distribution.

After that the prescriptive recommendations should be ignored. Informational recommendations are useful in a longer term. They describe significant properties of FA, which are hard to obtain automatically and are used to construct better distribution by knowing the particular case and thus using better particular distribution construction algorithms and heuristics. Once system algorithms of static and dynamic analysis become more powerful, informational recommendations become superfluous. At that point pure FA is sufficient to construct an efficient parallel program.

According to this approach FA is made free of all non-functional decisions, which include multiple assignment (data mutability), order of computations (except informational dependencies), resources distribution, garbage collection and so on. In Charm++, for instance, multiple assignment present, which is currently employed to optimize performance, but later it will become an obstacle for existing Charm++ programs. Recommendations currently play critical role in achieving high performance, because current knowledge in parallel programming automation is not enough to efficiently execute such high performance representations as FA automatically. Recommendations cover the lack of such knowledge and allow to achieve satisfactory performance of FA execution.

3 LuNA System

FP is described in two languages — LuNA and C++. LuNA is used to specify DFs and CFs, as well as recommendations, while C++ is used to define sequential subroutines, which are used to implement CFs in run time. C++ is a powerful conventional language, supported by well-developed compilers and other tools, thus making single jobs — CFs — highly efficient, leaving the system solely with problems of distributed execution.

Older LuNA releases employed the semi-interpretation approach, where FP is interpreted in run time by LuNA run-time system. With this approach the run-time system interprets FP, constructs internal objects, which correspond to CFs and DFs, distributes them to computing nodes, transfers input DFs to CFs and executes CFs once all input DFs are available locally, etc. Current LuNA release employs conceptually the same, but practically more efficient approach. With this approach each CF is considered as a lightweight process, controlled by a program and being executed in a passive run-time environment, accessible via API. Program for each CF is generated automatically by LuNA compiler and usually comprises the following main steps:

- Migrate to another node (if needed), where CF will be executed,
- Request input DFs and wait for them to arrive,
- Perform execution on input DFs with production of output DFs,
- Spawn and initialize new CFs,
- Perform finalization actions.

Finalization actions may include deletion of DFs, storing computed DFs to current or remote computing nodes and so on. Certain steps may vary depending on CF type (single CF execution, subroutine invocation, for- or while- loop, if-then-else operator, etc.), allowed in LuNA language. (Here and below CF's program denotes the program, generated for the CF by LuNA compiler, which should be differentiated from C++ sequential subroutines, which are provided by user as a part of FP.) CF's program also depends on compiler algorithms, recommendations, hardware configuration, etc. Generally, all static decisions on how FP should perform are formulated as CFs' programs. Note, that CFs' programs are not rigid. For instance, the migration step is statically generated, but exact node and route to it may be computed dynamically. Generally, all dynamic decisions are left to run time.

Since CF's programs are generated in C++, they are also optimized by conventional C++ compiler, which takes care of many minor, but important optimizations, such as static expressions evaluation, dead code elimination, call stack optimizations and all other optimizations conventional compilers are good at.

While delegating serial code optimization (sequential CF's implementations and CF's generated programs) to a well-developed C++ compiler, LuNA compiler and run-time system focus on the distributed part of the execution. Based on recommendations, decisions on CFs and DFs distribution to computing nodes, order of CFs execution, garbage collection and others are made statically (in LuNA compiler) and/or dynamically (in run-time system). Consideration of these algorithms is out of scope of the paper and can be found in other publications on LuNA system.

4 Performance Evaluation

To investigate performance of generated programs in comparison with the previous approach a number of tests was conducted. As an application a model 3D heat equation solution in unit cube is considered. This application was studied in our previous paper [13], 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 [14].

The testing was conducted on MVS-10P supercomputer of the Joint Supercomputer Centre of Russian Academy of Sciences [15]. 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: from 2^3 (8) to 6^3 (216) with step 1 (in each dimension).

The results are shown in Fig. 1. Here LI (LuNA-Interpreter) denotes the previous LuNA release, where run-time interpretation approach is employed, while LC (LuNA-Compiled) denotes the current approach, where CFs' programs are generated. MPI denotes the reference implementation, hand-coded using Message Passing Interface.

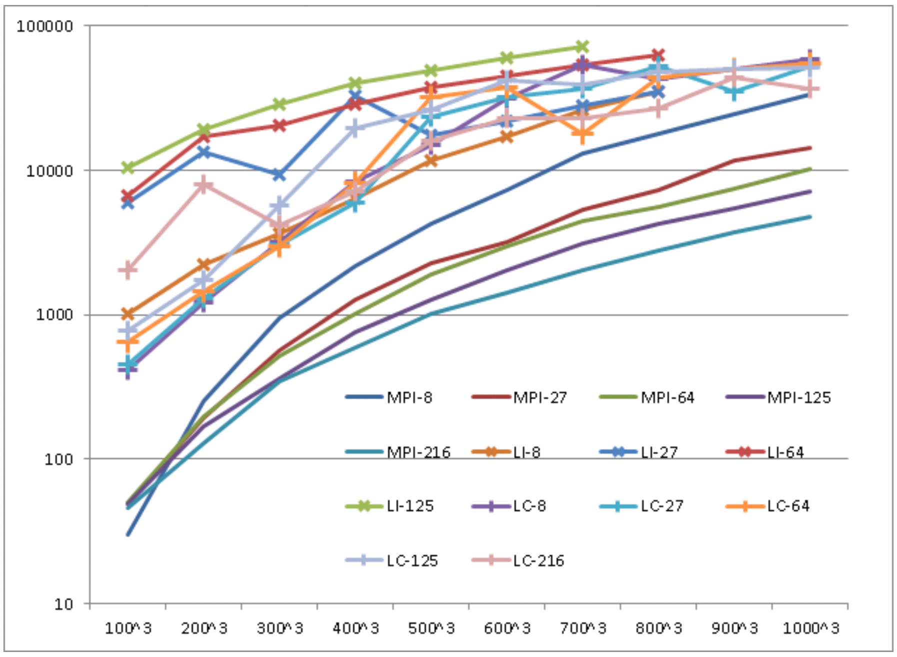

Fig. 1. Program execution time (in seconds). MPI- LI- and LC- are MPI-based, LuNA-Interpreter and LuNA-Compiled implementations correspondingly. The number denotes the number of cores. The X axis is the mesh size.

From Fig. 1 it can be seen, that current LuNA release produces a much more efficient implementation, than the previous release, although reference MPI implementation outperforms them both. It also can be seen, that the most advantage LC over LI can be observed for smaller fragments sizes, which is expected, since serial code optimization mainly reduces overhead, which is proportional to number of fragments (and not their sizes, for example). The reference MPI implementation is about 10 times faster, which means, that more optimizations are required. In particular, network overhead, imposed by run-time system communications, has to be reduced. However, such a slowdown may be tolerable, because, firstly, development of FP required less skill and effort from the user, and, secondly, with system optimization existing FPs become more efficient as a consequence without any need to change.

5 Conclusion

An approach to achieve efficient execution of parallel programs, defined in a high level language, is considered, as well as its implementation in LuNA system for automated parallel programs construction. Performance tests were conducted to compare current LuNA performance with the previous release and reference hand-coded implementation of the same test. In the future both software optimization and development of intelligent system algorithms are required to achieve better performance.

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. Valkovsky, V., Malyshkin, V.: Synthesis of parallel programs and systems on the basis of computational models. Nauka, Novosibirak (1988).
13. Akhmed-Zaki, D., Lebedev, D., Perepelkin, V. J Supercomput (2018). https://doi.org/10.1007/s11227-018-2710-1
14. Sapronov, I., Bykov, A.: Parallel pipelined algorithm. Atom 2009, no 44, pp 24–25 (2009) (in Russian)
15. Joint Supercomputing Centre of Russian Academy of Sciences Official Site, http://www.jscc.ru/, last accessed: 2019/04/01