Trace-Based Optimization of Fragmented Programs Execution in LuNA System

Victor Malyshkin[1,2,3][0000-0002-7874-3686] 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
perepelkin@ssd.sscc.ru

Abstract. Automatic construction of high performance distributed numerical simulation programs is used to reduce complexity of distributed parallel programs development and to improve code efficiency as compared to an average manual development. Development of such means, however, is challenging in general case, that's why a variety of different languages, systems and tools for parallel programs construction exist and evolve. Program tracing (i.e. journaling execution acts of the program) is a valuable source of information, which can be used to optimize efficiency of constructed programs for particular execution conditions and input data peculiarities. One of the optimization techniques is trace playback, which consists in step-by-step reproduction of the trace. This allows reducing run-time overhead, which is relevant for runtime system-based tools. The experimental results demonstrate suitability of the technique for a range of applications.

Keywords: Automatic Program Construction, Fragmented Programming Technology, LuNA System, Trace Playback.

1 Introduction

Development of high performance scientific parallel programs for supercomputers is often complicated and hard due to the necessity to decompose data and computations, organize parallel data processing, provide non-functional properties of the programs. Such properties may include efficiency (execution time, memory consumption, network load, etc.), static or dynamic workload balancing, fault tolerance, checkpointing, etc. All this requires in-depth knowledge of hardware architecture, skill in parallel programming, familiarity with appropriate parallel programming methods and tools. This makes manual programming troublesome for an average supercomputer user, who is an expert in the subject domain, not in system parallel programming. Usage of parallel programming automation systems, languages and tools allows to significantly reduce complexity of parallel programming, improve quality of produced programs and reduce knowledge and skill requirements a programmer has to possess.

In general automatic construction of an efficient parallel program is algorithmically hard, which why no effective general approach is expected to exist, so a diversity of various approaches, heuristics, languages and programming systems are being constantly developed to support parallel programming automation in different particular cases and subject domains. One of the promising approaches of parallel programs automatic optimization is trace-based optimization. This approach assumes that a program is first run on some characteristic input data, and its performance is being recorded as a trace of events (computational, communicational, etc.). The trace is then analyzed to extract quantitative information and pass it to a programming system (compiler, interpreter, etc.) to produce more efficient code. This is similar to profile-based optimization, except that a profile contains statistical information, while a trace contains the full log of significant events. In particular, trace can be used to reproduce the computation process recorded ("trace playback"), which can be more efficient than the normal program execution if the latter involves dynamic decision-making or other overhead, which can be omitted with trace playback. This, however, is not always possible, because change in input data or the computing system state may cause inconsistent execution. This paper is devoted to implementation of this idea in LuNA system for distributed parallel programs construction [1].

The rest of the paper is organized as follows. Section 2 contains a brief necessary introduction into LuNA system computational model in comparison with other systems. Section 3 describes how trace gathering and playback are implemented in LuNA. Section 4 presents results of the experimental study.

2 Trace Playback in LuNA System

2.1 LuNA System

LuNA (Language for Numerical Algorithms) is a language and a system for automatic construction of numerical distributed parallel programs for distributed memory parallel computers (multicomputers). It is an academic project of the Institute of Computational Mathematics and Mathematical Geophysics of the Siberian Branch of Russian Academy of Sciences. The system is based on the theory of structured synthesis of parallel programs [2], and its purpose is to support the active knowledge technology [3]. LuNA program is a high-level coarse-grained explicitly-parallel description of a numerical algorithm, which is basically a description of a bipartite oriented graph of computational fragments (CFs) and data fragments (DFs). DF is an immutable piece of data, the result of data decomposition. Each CF is a conventional subroutine call, which takes a number of DFs as inputs to compute values of a number of other DFs (these production-consuming relations correspond to arcs in the graph). So, LuNA program defines a set of informationally dependent tasks (CFs), which have to be executed in an order, which satisfies the dependencies. To execute such program LuNA has to distribute CFs to computing nodes, perform DFs transfer from producers to consumers and execute CFs after all their input DFs are available at the node. Efficiency of such execution is conditioned by the CFs distribution and execution order, by DFs network transfer delays and by the run-time system overhead. As our previous works show [4–11] the performance of LuNA programs is 1-100 times less than that of manually developed programs, depending on the subject domain. We continue to improve LuNA system algorithms to provide better performance for practical application classes, and this work is one of such improvements. More details on LuNA system can be found in [1] and in its public repository[1].

2.2 Trace Playback in LuNA System

Since LuNA program execution consists eventually of CFs executions, the trace information includes CF execution start and end times and the node on which the CF was executed. This information is sufficient to completely reproduce the computation of LuNA program. Once the trace is recorded, its playback on each computing node may be organized as follows:

1. Pick the earliest unexecuted CF *a* from the trace (on the node).
2. For each input DF *x* of the CF *a* find in the trace the CF *b*, which produced it.
3. If CF *b* was executed on the same computing node where CF *a* was executed, then DF *x* is available on the node; otherwise receive DF *x* as a message from CF *b*'s node.
4. Invoke the conventional subroutine, related to CF *a* with input DFs passed to it.
5. For each output DF *x* of the CF *a* find all CFs *c*, which take DF *x* as input. If CF *c* is located on the same node as CF *a*, then store DF *x* locally, otherwise send DF *x* as a message to CF *c*'s node.

This is an essential scheme, although some more or less obvious tuning should be done in practical implementation. For example, if multiple CFs are located on the same computing node and take the same DF as input, then only one copy of the DF should be passed via network. Note that trace playback can be performed in multiple threads for each node (normal LuNA operation is also multi-threaded on each computing node).

This scheme misses the garbage collection, which takes place with normal LuNA operation. It can be straightforwardly implemented by recording to the trace the relative time point where DF deallocation took place. However, this appeared to be redundant, since all actual DFs consumptions are explicitly seen in the trace, thus the DF deallocation is performed as soon as last consumption on the node has occurred.

With trace playback the run-time overhead is reduced to the minimum. No decision making on CFs distribution, CFs execution ordering or DFs garbage collection is needed. In particular, LuNA dynamically balances workload by redistributing CFs to computing nodes, but only a final location where execution took place matters. All multi-hop DF transfers become single-hop transfers. Reduction of most kinds of overhead is the main source of performance improvement for trace playback as compared to normal LuNA operation.

Note, that LuNA programs execution is non-deterministic in sense of CFs distribution and execution ordering, and in sense of timings. Even minor factors (such as

[1] https://gitlab.ssd.sscc.ru/luna/luna

network delays or external CPU load) may influence the decisions LuNA system makes and implements. Dynamic load balancing is especially sensible to such factors. The trace, however, is much more deterministic, since most events are rigidly fixed.

3 Discussion and Related Works

3.1 Analysis and Discussion

The main drawback of the approach is that the set of CFs may depend on input data. As long as the task graph (the set of CFs) persists trace playback produces valid execution for any input data. But, for example, if the number of loop iterations depends on input data, then trace playback may be erroneous. This drawback can be partially compensated by two factors. First, there are a lot of applications where tasks graph does not depend on input data (e.g. dense linear algebra operations). Second, the fact that the task graph appeared to be different for given input can be detected automatically. In particular, in LuNA there are three operators, which can produce data-dependent task graphs: `if`, `for` and `while`. Each of the operators can be supplied with straightforward checks, which will ensure that each `if` condition was resolved to the same `true`/`false` value and that every `for` and `while` operator has the same iteration range. So, trace playback engine can inform the user on unsuccessful playback (rather than silently perform erroneous execution) and suggest normal program execution.

To some extent this drawback can be overcome further. E.g. some kind of induction techniques can be used to stack `for` or `while` loop iterations into a parametric range-independent form. For the `if` operator both *then* and *else* branches can be traced at first precedent, and after that the execution of both branches can be done via trace playback. Study of these possibilities is out of the scope of the paper.

Another drawback of the approach is that no decisions on CFs distribution and execution order are made – only the decisions made by LuNA system in the traced run are recorded and reproduced. These decisions may be not good for a number of reasons. E.g., hardware configuration or its external load may be different; CF execution time may depend on input data; absence of run-time system overhead may influence timings, etc. The decisions themselves, that LuNA system has made, can be not good, because LuNA system algorithms are not perfect. This brings us to the idea of trace optimization and tuning before doing the playback. Study of the idea is beyond the current work, but a brief overview of the problem can be given. Firstly, the trace can be analyzed for work imbalance or inefficient CFs execution order. Secondly, any CF can be reassigned to another node with no risk of bringing error to the execution. Also, CFs execution can be reordered unless informational dependencies are violated. Such trace transformations can either eliminate work imbalance or retarget the trace to another hardware configuration (computing nodes number, network topology, relative nodes performance, etc.).

Besides trace optimization, trace execution engine can be improved. For example, the above mentioned thread pool-based execution is one of such possible optimizations. More dynamic improvements can be made. For example, dynamic workload balancing may be employed to eliminate work imbalance that occurs during trace playback. Study of these possibilities is also out of the scope of the paper.

3.2 Related Works

Trace playback is practical in LuNA system because of the computational model it employs. In particular, the “computational” part of a LuNA program is separate from distributed management logic, which allows to replace the latter with a trace play-back engine. There are other programming languages and systems, where trace play-back may be of use. For manually developed conventional distributed parallel pro-grams trace playback appears to be inapplicable, since it is impossible to distinguish the essential computational part from the rest of the code, which organizes parallel computations, communications and data storage.

Specialized computational models, such as the map-reduce model [12,13], allow to distinguish the computational part and computations structure since it is explicitly formulated in the code. This, in turn, allows to trace execution and playback the trace. For example, such systems implement dynamic workload balancing, which causes some run-time overhead. It can be reduced by the trace playback technique. Of course, this makes sense for a series of computations where imbalance is known to be the same. The rest of overhead is usually negligible.

Task-based systems, such as Charm++ [14] or OpenTS [15] allow trace playback mostly the same way it is possible in LuNA system. In Charm++, however, it may be harder to implement, because *chares* (Charm++ decomposition units) may behave differently depending on the order in which they receive messages from other chares. To allow safe trace playback some additional constraints to chare codes may be re-quired.

For systems with explicit program behavior control, such as PaRSEC [16] or Le-gion [17] trace playback seems to be as easily implemented as in LuNA, since the computational part is explicitly formulated in the computational model.

Some possibilities of trace playback exist in systems for automated serial code par-allelization, such as DVM-H [18]. Here a serial code is annotated (either manually or automatically) with “parallelization pragmas”, and a parallel program is generated automatically. In particular, a dynamic workload balancing mechanism may be in-cluded into the generated program. Code annotations allow identifying the computa-tional part, and since the distributed code is generated automatically, it can be instru-mented to trace events, necessary to perform the playback.

It can be concluded, that trace playback is a reasonable technique for programming languages, systems and tools, which employ run-time systems, or at least provide some dynamic properties (such as dynamic load balancing) at cost of some overhead.

4 Experiments

To playback a trace a series of actions to perform is generated for each computing node. Possible actions are invocation of a serial subroutine, DF transfer to another node and DF deletion. Such series is easily constructed from trace. Implementation of the series of actions on each node causes the trace playback.

The naïve way to implement trace playback is to generate the series of actions as a conventional (e.g. MPI-based) program. Such an approach possesses minimal possible overhead. In practice, however, such source code listing grows large and takes too much time to be compiled into binary (e.g. hours of compilation for a large program). To overcome this issue the series of events was encoded into a binary file (to reduce size), and a trivial interpreter was developed, which decodes the file and performs the actions using a worker thread pool on each node. This decoding does add some overhead, but it is usually negligible due to coarse granularity of CFs. A separate thread was dedicated to receiving messages from other nodes.

For experimental performance evaluation a Particle-In-Cell application for self-gravitating dust-cloud simulation [19] was used as an example of a rather complicated real supercomputing-targeted application. Tests were conducted on MVS-10p cluster of Joint Supercomputing Center of Russian Academy of Sciences[2]. The testing was conducted for various parameters (see Table 1) to investigate performance in different conditions.

Table 1. Experimental results

Parameters			**Execution time (sec.)**		
Mesh size	**Particles**	**Cores**	**MPI**	**LuNA-TB**	**LuNA**
100^3	10^6	64	5.287	13.69	355.5
150^3	10^6	64	18.896	31.088	732.833
150^3	10^7	64	23.594	111.194	2983
150^3	10^7	125	23.352	118.32	3086.411
150^3	10^6	343	33.697	39.651	1008.655

Two programs, developed by S. Kireev for [20], which implement the same algorithm, were used. The first program is a conventional C++ distributed parallel program, based on Message Passing Interface (MPI). The second one is a LuNA program. Execution time for these programs is shown in Table 1. There is also a column labeled LuNA-TB. This is the execution time of the same LuNA program, but using the trace-playback technique. The MPI program can be considered as a reference point, its efficiency is what one can expect as a result of manual development of an experienced applied programmer. The LuNA program is an automatically constructed program using a general approach, and its efficiency is expectedly much lower, than that of the MPI program. And the LuNA-TB is somewhere in the middle, an automatically generated program using the particular trace playback approach.

The main result of the testing is that LuNA-playback indeed significantly speeds up execution of LuNA programs. This confirms that the trace playback is a useful technique for optimizing efficiency of automatically constructed parallel programs. Its efficiency is still lower than that of the MPI program, but this is obviously a practically usable result, considering that it is obtained automatically.

[2] http://www.jscc.ru

It can also be seen from Table 1 that trace playback approach is more advantageous for programs with finer granularity, where fragments are of lesser size. The advantage is the bigger the more computing nodes are involved in computation. This is also expected, since dynamic decentralized algorithms employed in LuNA produce significant overhead, which is cut off with trace playback.

5 Conclusion

The trace playback technique is investigated as a distributed programs optimization technique for parallel programming automation systems. Trace playback was implemented for LuNA system for automatic numerical parallel programs construction. The experiments showed a significant improvement of the efficiency of constructed programs. Possible improvements of the technique, aimed at overcoming its drawbacks are briefly discussed. It can be concluded that the trace playback technique is practical for high performance distributed parallel programs construction automation, which can be used automatically (along with other particular system algorithms and heuristics). In future we plan to further investigate the approach within LuNA system to widen the application class this technique is applicable to.

The work was supported by the budget project of the ICMMG SB RAS No. 0251-2021-0005.

References

1. Malyshkin V.E., Perepelkin V.A. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem. In: Parallel Computing Technologies. PaCT 2011. Lecture Notes in Computer Science, vol 6873. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-23178-0_5 (2011)
2. V.A. Valkovsky, V.E. Malyshkin.: Synthesis of Parallel Programs and Systems on the Basis of Computational Models (In Russian). Nauka, Novosibirsk (1988)
3. Malyshkin V. Active Knowledge, LuNA and Literacy for Oncoming Centuries. In: Bodei C., Ferrari G., Priami C. (eds) Programming Languages with Applications to Biology and Security. Lecture Notes in Computer Science, vol 9465. Springer, Cham. https://doi.org/10.1007/978-3-319-25527-9_19 (2015)
4. 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.
5. Akhmed-Zaki, D., Lebedev, D., Perepelkin, V. Implementation of a 3D model heat equation using fragmented programming technology // J Supercomput. 2019. pp. 7827-7832. DOI: 10.1007/s11227-018-2710-1.
6. B. Daribayev, V. Perepelkin, D. Lebedev, D. Akhmed-Zaki. Implementation of the Two-Dimensional Elliptic Equation Model in LuNA Fragmented Programming System // 2018 IEEE 12th International Conference on Application of Information and Communication Technologies (AICT). 2018. pp. 1-4.
7. Nikolay B., Perepelkin. V. Automated GPU Support in LuNA Fragmented Programming System // Parallel Computing Technologies. PaCT 2017. Lecture Notes in Computer Sci-

ence, vol 10421.. Springer, Cham, 2017. pp. 272-277. DOI: 10.1007/978-3-319-62932-2_26.

8. Malyshkin. V., Perepelkin. V., Schukin G. Scalable Distributed Data Allocation in LuNA Fragmented Programming System // Journal of Supercomputing, S.I.: Parallel Computing Technologies - 2017. Springer, 2017. pp. 1-7. DOI: 10.1007/s11227-016-1781-0.
9. V.E. Malyshkin, V.A. Perepelkin, A.A. Tkacheva. Control Flow Usage to Improve Performance of Fragmented Programs Execution // In Proc 13th International Conference on Parallel Computing Technologies. LNCS 9251. Springer, 2015. pp. 86-90. DOI: 10.1007/978-3-319-21909-7_9.
10. Victor E. Malyshkin, Vladislav A. Perepelkin, Georgy A. Schukin. Distributed Algorithm of Data Allocation in the Fragmented Programming System LuNA // In Proc 13th International Conference on Parallel Computing Technologies. LNCS 9251. Springer, 2015. pp. 80-85. DOI: 10.1007/978-3-319-21909-7_8.
11. Norma Alias, Sergey Kireev. Fragmentation of IADE Method Using LuNA System // PaCT-2017 proceedings, LNCS 10421, Springer, 2017, P.85-93. DOI: 10.1007/978-3-319-62932-2_7
12. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters / OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004. p 137-150.
13. Tom White. Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale / O'Reilly Media; 4th edition (April 21, 2015), 756 p. ISBN-13 : 978-1491901632
14. Kale, Laxmikant V. and Bhatele, Abhinav. Parallel Science and Engineering Applications: The Charm++ Approach / Taylor & Francis Group, CRC Press. 2013. ISBN 9781466504127
15. Moskovsky A., Roganov V., Abramov S. (2007) Parallelism Granules Aggregation with the T-System. In: Malyshkin V. (eds) Parallel Computing Technologies. PaCT 2007. Lecture Notes in Computer Science, vol 4671. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-73940-1_30
16. George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack Dongarra. 2013. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. Computing in Science and Engineering 99 (2013), 1.
17. Michael Bauer, Sean Treichler, Elliott Slaughter and Alex Aiken. Legion: expressing locality and independence with logical regions // onference on High Performance Computing Networking, Storage and Analysis, SC'12, Salt Lake City, UT, USA, November 11 - 15, 2012. 10.1109/SC.2012.71
18. N.A. Kataev, A.S. Kolganov. The experience of using DVM and SAPFOR systems in semi automatic parallelization of an application for 3D modeling in geophysics // The Journal of Supercomputing, US: Springer, 2018, P. 1-11
19. Kireev S. A parallel 3D code for simulation of self-gravitating gas-dust systems. InInternational Conference on Parallel Computing Technologies 2009 Aug 31 (pp. 406-413). Springer, Berlin, Heidelberg.
20. Belyaev N., Kireev S. (2019) LuNA-ICLU Compiler for Automated Generation of Iterative Fragmented Programs. In: Malyshkin V. (eds) Parallel Computing Technologies. PaCT 2019. Lecture Notes in Computer Science, vol 11657. Springer, Cham. https://doi.org/10.1007/978-3-030-25636-4_2