High-Efficiency Specialized Support for Dense Linear Algebra Arithmetic in LuNA System

Nikolay Belyaev[1[0000-0002-8455-0995]] 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
perepelkin@ssd.sscc.ru

Abstract. Automatic synthesis of efficient scientific parallel programs for supercomputers is in general a complex problem of system parallel programming. Therefore various specialized synthesis algorithms and heuristics are of use. LuNA system for automatic construction of distributed parallel programs provides a basis for accumulation of such algorithms to provide high-quality parallel programs generation in particular subject domains. If no specialized support is available in LuNA for given input, then the general synthesis algorithm is used, which does construct the required program, but its efficiency may be unsatisfactory. In the paper a specialized run-time system for LuNA is presented, which provides runtime support for dense linear algebra operations implementation on distributed memory multicomputers. Experimental results demonstrate, that automatically generated parallel programs of the class outperform corresponding ScaLAPACK library subroutines, which makes LuNA system practically applicable for generating high performance distributed parallel programs for supercomputers in the dense linear algebra application class.

Keywords: Parallel Programming Automation, Fragmented Programming Technology, LuNA System, Distributed Dense Linear Algebra Subroutines.

1 Introduction

This paper is devoted to the problem of efficient parallel program construction automation in the field of high performance scientific computations on supercomputers. Efficiency is a mandatory requirement for such programs. Otherwise costly high performance computing resources are wasted. Provision of efficiency of a parallel program is a hard problem (NP-hard in general case), which makes such program construction automation challenging. The complexity of efficiency provision arises from the necessity to decompose data and computations and organize parallel data processing in such a way that as much of hardware resources as possible are loaded fully and evenly with useful computations. Manual development of efficient parallel programs requires knowledge of distributed hardware architecture, familiarity with methods and tools for distributed parallel programming, skills in system parallel programming. Such expertise is different from the expertise in the subject domains, to

which computations are related. Manual parallel programs development compels users to possess expertise in both domains. This conditions the importance of program construction automation tools, which allow one to describe computations with a higher level programming language (or an API), and expect an efficient parallel program to be constructed and executed automatically. Such an approach allows encapsulating much of the expertise a parallel programmer needs to possess into a programming system and automatically apply the encapsulated knowledge for program construction. Since no general solution exists, of practical interest are particular and heuristic solutions, capable of providing satisfactory efficiency for certain application classes. Also of practical interest are approaches, aimed at accumulation and automatic application of various particular solutions.

Nowadays the need in parallel programming automation means tends to increase, since supercomputers' hardware and software grow more complex. Heterogeneity of hardware increases, number of nodes and cores per node increases, network and memory subsystems become more lagging behind cores and therefore more critical, co-processors usage becomes essential to maximize performance, etc. Taking all this into account is both necessary and hard, so research in the field of parallel programming automation is more and more demanding.

Many programming systems, languages and tools exist and evolve to assist or replace programmers [1,2].

Charm++ [3,4] is an open-source parallel system which consists of distributed runtime system which is able to execute a distributed computational tasks (chares) graph on a supercomputer. Each task is able to communicate with others by sending and receiving messages. An applied programmer has to program communications between tasks by hand using low-level C++ interface. The task-based computational model, employed in Charm++ allows using particular system algorithms to support various classes of applications, but in general the peculiarities of the model make Charm++ programs partially opaque to the system because of low-level message passing means employed. That impedes Charm++'s capability to accumulate particular system algorithms.

PaRSEC [5] is a parallel programming system, designed specifically for automated generation of efficient parallel programs, which implement linear algebra operations. An applied programmer describes a tasks graph using the built-in high level language. This simplifies the process of development of high performance parallel programs. PaRSEC is able to generate programs only for the restricted class of linear algebra algorithms.

Legion [6], Regent [7–9] and LuNA [10] systems are also able to execute an algorithm described as a task graph on a supercomputer. These systems use general system algorithms to distribute tasks to computing nodes and execute the graph. The systems also provide powerful means to provide specialized support of program construction and execution, because execution control algorithms are excluded from the algorithm description, thus program construction and execution can be varied freely to support efficient execution of applied algorithms in particular subject domains. The systems are therefore suitable for accumulating various system algorithms for different subject domains.

It can be seen that a great effort is being put into automating programming. It is also clear that the efficiency problem is far from being solved for many subject domains.

In the presented work we employ LuNA as the system capable for particular system algorithms accumulation. LuNA is a system for automatic construction of scientific parallel programs for multicomputers. It is an academic project of the Institute of Computational Mathematics and Mathematical Geophysics of the Siberian Branch of Russian Academy of Sciences. This system is aimed at automatic construction of high performance distributed parallel programs for conducting numerical computations on supercomputers. It focuses on providing to a user an ability to describe computations, that need to be conducted, in a high-level platform-independent form. Also it provides some high level means (called recommendations) to express a programmer idea on how to organize efficient parallel execution on a supercomputer. This approach is based on the structured synthesis theory [11] and conforms to the active knowledge technology [12]. It allows to significantly reduce the complexity of efficient parallel program generation problem without the need for the programmer to do low-level parallel programming. Source code of LuNA system can be found in its public repository[1].

In this paper we investigate how satisfactory efficiency can be achieved in LuNA by making a specific system support for a particular subject domain, namely, dense linear algebra operations. This support is implemented as a particular run-time system, which is capable of execution of LuNA programs (or subprograms) of particular form, common for many dense linear algebra operations. The run-time system takes into account peculiarities of the operations to achieve high efficiency, comparable with that of ScaLAPACK, which is a widely used library for such operations. This demonstrates that LuNA system can be a useful tool for practical construction of high performance scientific programs for subject domains, reasonably supported by specialized system algorithms.

The rest of the paper is organized as follows. Section 2 describes the proposed approach to support dense linear algebra operations in LuNA. Section 3 presents the experimental results, where LuNA performance is compared to that of ScaLAPACK on some operations. Conclusion ends the paper.

2 Particular Execution Algorithms Approach

2.1 Main Idea

This section describes the overall idea of the proposed solution. For the class of numerical algorithms particular distributed run-time system algorithms are developed and integrated to LuNA system. LuNA system analyzes the input algorithm description written in LuNA language and determines the class to which the input algorithm description belongs. Then LuNA compiler selects particular system algorithms which are used to automatically generate a parallel program by the input algorithm descrip-

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

tion. The result of compilation is a C++ code, which can be compiled by a conventional C++ compiler and linked against a library, which implements the run-time system. Then it is able to be executed on a supercomputer.

In this paper only a single class of numerical algorithms is considered to demonstrate the approach. This class contains widely used matrix algorithms such as LU, LL^T, LDL^T and similar matrix factorization algorithms. One of the advantages of the approach is that it is possible to identify whether input algorithm belongs to the class or not. No sophisticated information dependencies analysis is required for that.

2.2 Main Definitions and Class of Algorithms Description

For further discussion the model of algorithm is described as it is one of the most important things when developing parallel programming systems. Firstly let some formal definitions be given. Secondly, the main idea of the model is given. Then the class of algorithms is described formally.

Definition 1. A **data fragment** (DF) is a the following tuple: $\langle N, V \rangle$, where N is a name (a regular string), V is an arbitrary value.

Definition 2. **DFs array** is the following set: $\{x | x = \langle h_1, \ldots, h_N \rangle, df_{\langle h_1, \ldots, h_N \rangle}, \forall i \in \{1, \ldots, N\}: 0 \leq h_i < M_i, h_i \in \mathbb{N}_0\}$, where $df_{\langle h_1, \ldots, h_N \rangle}$ is a DF, M_i is the size of i-th dimension of the array, $\langle h_1, \ldots, h_N \rangle, N \in \mathbb{N}, \forall i \in \{1, \ldots, N\}: 0 \leq h_i < M_i$ is a tuple of array element indices.

Definition 3. Let concept of **task argument** now be defined as follows:

1. Every DF is a task argument
2. The following tuple is a task argument: $\langle A, \langle h_1, \ldots, h_N \rangle \rangle$, where A is an N-dimensional array of DFs. This kind of argument is also called **array-argument**.

Definition 4. A **task** is the following tuple: $\langle n, I, O \rangle$, where n – name (regular string), $I = \{a_1, \ldots, a_M\}, M \in \mathbb{N}_0$- set of task arguments called **input arguments**, $O = \{b_1, \ldots, b_K\}, K \in \mathbb{N}_0$ - set of task arguments called **output arguments**.

Definition 5. **Algorithm** is a tuple $\langle A, D, T \rangle$, where A – is a finite set of DFs arrays, D – is a finite set of DFs, T – is a set of tasks.

Let the main idea of the algorithm model now be explained. An applied programmer describes the data processed by a numerical algorithm with a set of DFs and DFs arrays (the description is the input for the system). Each DF is associated with a value which may store arbitrary data. For example, the value of some DF may store a dense matrix block, a vector part or a single value of some type. For each DFs array the applied programmer provides a mapping function. The mapping function maps DFs array elements to some memory location depending on array element indices and the computing node to which an array element is distributed. Many DFs array elements may be mapped to the same memory location. In this case, the applied programmer is responsible for avoiding collisions, i.e. when different elements, mapped to the same location, are in use at the same time span. Then the applied programmer describes a set of tasks. Each task transforms the values of its input DFs to the values of its output

DFs by calling associated external routine. Such routine is implemented by the applied programmer with some conventional language, such as C, C++ or Fortran. For example, there may be implemented an external routine that multiplies two dense matrix blocks, represented as DFs. Multiple tasks may be associated with the same external routine. At run-time this external routine with the values of the input DFs forms a task that can be executed by LuNA system when all values of the input DFs are computed. After the task is executed the values of its output DFs become computed, so some other tasks may become executable.

Consider now a class of algorithms that is handled by the developed particular system algorithms. The class of algorithms consists of algorithms that meet the following requirements:

1. Every task within the algorithm has either only one output array-argument or all array element indices of all output array-arguments of the task are pairwise equal.
2. Dimension of all DFs arrays is the same, and the sizes of each dimension are pairwise equal.

For example, Cholesky (LL^T) factorization algorithm mentioned above meets the requirements. Also this class contains many other matrix algorithms such as LU factorization, LDL^T factorization and others.

2.3 Compiler

One of the important components of the developed LuNA system extension is a compiler that checks if the input algorithm description belongs to the supported class of algorithms. If the input numerical algorithm meets the above requirements the compiler generates a parallel program according the following principle. For each task described within the input algorithm description compiler generates a C++ lambda function (it is called run-time task). The body of the lambda-function consists of a C++ call statement of the routine associated with the task. Then compiler generates a call to the run-time library that implements distributed execution of the input algorithm (the execution algorithm is described in Section 2.4). This call submits the task to the executor.

At run-time a set of tasks with their arguments forms a bipartite directed acyclic tasks graph (DAG) which is submitted to a distributed executor implemented in the run-time library. The executor distributes DFs and DFs arrays to nodes and asynchronously executes the tasks graph on the multicomputer. Fig. 1 shows the overall structure of generated program.

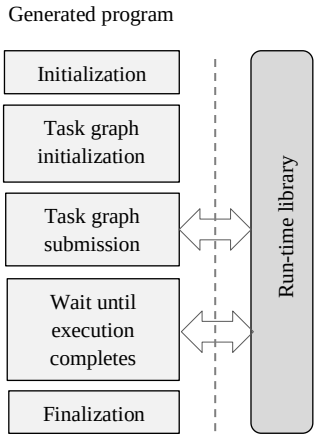

Fig. 1. Structure of generated program.

2.4 Run-Time Library and Task Graph Execution

Consider now the distributed tasks graph executor that is implemented in the run-time library. At first, consider the data distribution algorithm. The value of each DF (not an DFs array element) is stored in the memory of all computing nodes. Each DFs array is distributed according to the block-cyclic [13] principle. The parameters of the block-cyclic distribution may be set by the applied programmer. The dimension of the block-cyclic distribution is equal to the dimension of the DFs arrays declared in the input algorithm description.

Consider now the principle of tasks mapping to computing nodes and execution of the tasks graph. At run-time each task is mapped to the computing node to which its output DFs arrays elements is mapped (according the requirements indices of all output array-arguments of a task are pairwise equal and thus all corresponding DFs array elements are mapped to the same computing node). If an input argument of a task is mapped to a different node, an asynchronous message is sent after the producer task execution. In addition, each computing node runs a receiver loop in a dedicated thread. When some task argument value is received, corresponding consumer tasks are found. When the values of all input arguments of a consumer task are obtained, it is executed. The process continues until all tasks are executed.

3 Performance Evaluation

To measure the performance of the implemented extension of LuNA system, a test implementation of the Cholesky factorization of a dense matrix was developed with LuNA language. Such factorization is an example of an algorithm with complex structure and information dependencies. For performance evaluation the same test

was implemented using a ScaLAPACK [14] implementation of Cholesky factorization. ScaLAPACK is a widely used library, where Cholesky factorization is implemented. Execution times of both implementations were compared. Both implementations used two-dimensional block cyclic distribution of the input matrix into square matrix of square blocks, and the block size was a parameter. OpenBLAS library (version 0.3.15) [15] implementation of BLAS and LAPACK subroutines was used for both tests. Both implementations used right-looking blocked Cholesky factorization algorithm [16].

Two square dense double-precision matrices of sizes 32768 and 65536 were used as input data. For each of the matrices a number of experiments were conducted using different matrix block sizes ranging from 256 to 2048. Execution times of both tests were measured.

Testing was conducted on MVS-10P cluster of the Joint Supercomputing Centre of Russian Academy of Sciences[2] on a two-dimensional grid of 2×2 computing nodes. Each node contains 32 cores and 16 GB of memory. All 32 cores of each CPU were used in all tests.

Fig.2 shows execution times comparison of the ScaLAPACK and the LuNA implementations for the input matrix of sizes 32768 (left) and 65536 (right).

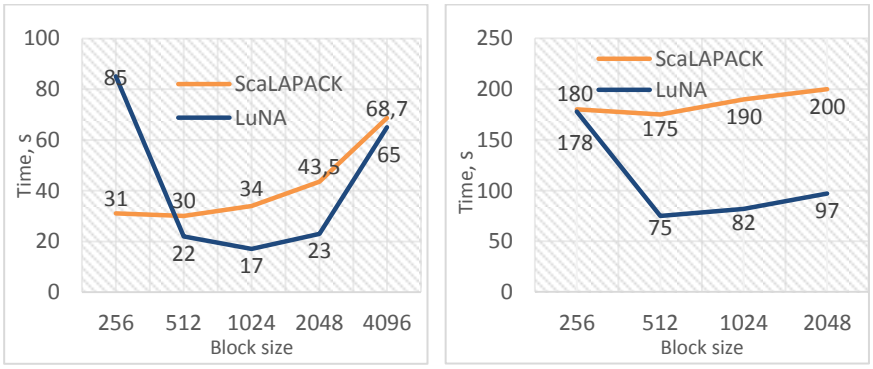

Fig. 2. Performance evaluation result for a square matrix of 32768 (left) and 65536 (right) elements.

Here in both cases the LuNA implementation outperforms the ScaLAPACK implementation of Cholesky factorization (by 2.4 times for matrix of 32768 elements and block size of 512 and by 2.1 for matrix of 65536 elements and block size of 1024).

The above results demonstrate that LuNA is able to generate an efficient parallel program from an algorithm description with complex information dependencies. The performance of the generated parallel program is approximately 2 times better than that of library developed by experts (for the studied test).

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

4 Conclusion

Automatic construction of efficient parallel programs generally requires different construction algorithms for different subject domains. LuNA system is capable of accumulating such algorithms. This ability was demonstrated by adding specialized support for dense linear algebra operations class. The achieved performance is comparable with that of a widely used library ScaLAPACK. This makes LuNA a practical tool for automatic construction of high performance distributed parallel programs for the applications class. Other classes of applications can also be particularly supported in LuNA in order to improve performance of automatically constructed programs if programs, constructed by general LuNA algorithms are not efficient enough.

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

References

1. Sterling, T., Anderson, M., & Brodowicz, M. (2017). A Survey: Runtime Software Systems for High Performance Computing. Supercomputing Frontiers And Innovations, 4(1), 48-68. DOI: 10.14529/jsfi170103
2. Thoman, P., Dichev, K., Heller, T. et al. A taxonomy of task-based parallel programming technologies for high-performance computing. J Supercomput 74, 1422–1434 (2018). DOI: 10.1007/s11227-018-2238-4
3. Kale, L. V., & Krishnan, S. (1993, October). Charm++ A portable concurrent object oriented system based on C++. In Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (pp. 91-108).
4. Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., ... & Kale, L. (2014, November). Parallel programming with migratable objects: Charm++ in practice. In SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 647-658). IEEE.
5. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., & Dongarra, J. J. (2013). Parsec: Exploiting heterogeneity to enhance scalability. Computing in Science & Engineering, 15(6), 36-45.
6. Bauer, M., Treichler, S., Slaughter, E., & Aiken, A. (2012, November). Legion: Expressing locality and independence with logical regions. In SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (pp. 1-11). IEEE.
7. Slaughter, E., Lee, W., Treichler, S., Bauer, M., & Aiken, A. (2015, November). Regent: A high-productivity programming language for HPC with logical regions. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-12).
8. Slaughter, E. (2017). Regent: A high-productivity programming language for implicit parallelism with logical regions (Doctoral dissertation, Stanford University).
9. Torres, H., Papadakis, M., & Jofre Cruanyes, L. (2019). Soleil-X: turbulence, particles, and radiation in the Regent programming language. In SC'19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-4).

10. 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)
11. V.A. Valkovsky, V.E. Malyshkin.: Synthesis of Parallel Programs and Systems on the Basis of Computational Models (In Russian). Nauka, Novosibirsk (1988)
12. 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)
13. Hiranandani, S., Kennedy, K., Mellor-Crummey, J., & Sethi, A. (1994, July). Compilation techniques for block-cyclic distributions. In Proceedings of the 8th international conference on Supercomputing (pp. 392-403).
14. Choi, J., Dongarra, J. J., Pozo, R., & Walker, D. W. (1992, January). ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In The Fourth Symposium on the Frontiers of Massively Parallel Computation (pp. 120-121). IEEE Computer Society.
15. Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 BLAS. ACM Trans. Math. Softw. 35, 1, Article 4 (July 2008), 14 pages. DOI: 10.1145/1377603.1377607
16. Kurzak, J., Ltaief, H., Dongarra, J., & Badia, R. M. (2010). Scheduling dense linear algebra operations on multicore processors. Concurrency and Computation: Practice and Experience, 22(1), 15-44.