LuNA-ICLU Compiler for Automated Generation of Iterative Fragmented Programs

Nikolay Belyaev[1,2] and Sergey Kireev[1,2](✉)

[1] ICMMG SB RAS, Novosibirsk, Russia
kireev@ssd.sscc.ru
[2] Novosibirsk State University, Novosibirsk, Russia

Abstract. The work focuses on the application of Fragmented Programming approach to automated generation of a parallel programs for solving applied numerical problems. A new parallel programming system LuNA-ICLU applying this approach was introduced. The LuNA-ICLU compiler translates a fragmented program of a particular type written in the LuNA language to an MPI program with dynamic load balancing support. The application algorithm representation and the system algorithms used in the LuNA-ICLU system are described. Performance comparison results show a speedup compared to the previous implementation of the LuNA programming system.

Keywords: Fragmented programming technology · LuNA system · Parallel program generation · Dynamic load balancing

1 Introduction

The problem of efficient parallel implementation of numerical algorithms on supercomputers remains relevant since the advent of supercomputers. Previously, low-level programming of processes or threads with different memory models was mainly used [1]. In recent decades, the growing diversity and complexity of computing architectures and the need to raise the level of programming have made automation of solving system parallel programming problems increasingly important. A number of parallel programming systems was developed in order to simplify the development of parallel programs. An overview of modern parallel programming systems for supercomputers may be found in [2,3]. The following features may characterize them.

– Separation of the application algorithm description from its implementation. A special algorithm representation is usually developed to describe the application algorithm [4–11]. The representation is supported by an API based on an existing language [4–6] (or its extension [7]) or a DSL [8–11]. Efficient

Supported by the budget project of the ICMMG SB RAS No. 0315-2019-0007.

execution of the algorithm presented in this way is provided by special system software, a compiler and/or a distributed runtime system.
- Fragmented representation of an algorithm. The complexity of the automatic decomposition of the application algorithm in general case still makes it necessary to perform the decomposition manually. Thus, the algorithm must be represented in a fragmented form [4–11].

A common representation of an algorithm for many parallel programming systems is a set of tasks (fragments of computations) linked by data and control dependencies, forming a graph. The system software provides parallel execution of tasks, while satisfying the dependencies. The task graph can be defined statically [9,11], or be formed dynamically during the execution of the program [5,6,10]. The static representation of the task graph has the advantage that the entire structure of the graph is known before execution, which allows wider scoped compile-time optimizations. Examples of systems with static task graph representation are PaRSEC (DAGuE) [9,10] and LuNA [11]. Compared to PaRSEC, the LuNA language can represent a wider class of algorithms.

LuNA system is an implementation of Fragmented Programming technology being developed at the ICMMG SB RAS in Novosibirsk, Russia. Program in LuNA language (fragmented program) defines a potentially infinite data flow graph, built of single-assignment variables called data fragments (DFs) and single-execution operations called fragments of computation (CFs). Each DF contains one or a portion of application variables. CFs compute some DFs from others. There are two types of CFs in LuNA language: atomic and structured. Atomic CFs are implemented by C/C++ subroutines, while structured CFs are bipartite graphs of CFs and DFs. The LuNA language supports the following structured CFs: conditional CFs ("if" operator), indexed sets of CFs ("for" and "while" operators), and subprograms ("sub" operator). CFs' or DFs' names may contain an arbitrary number of indices, that allow them to be interpreted as arrays.

The current implementation of the LuNA runtime system is a distributed interpreter of LuNA programs. In the process of execution it gradually unfolds a compact notation of a potentially infinite task graph, performing dynamical management of a distributed set of DFs and CFs. However, the use of universal control algorithms in the implementation has led to the fact that the LuNA runtime system has a considerable overhead, which leads to a poor performance on real-world applications [12,13].

The paper presents another approach to the implementation of the LuNA system based on the static translation of a LuNA program into an MPI program. In this approach, the set of supported algorithms was narrowed to a class of iterative algorithms over rectangular n-D arrays, where n is the dimension of the array. The LuNA language was extended by additional high-level constructs in order to ease the program analysis. The implementation of this approach is a new LuNA-ICLU compiler. It provides construction of an MPI program with dynamic load balancing support. Using the example of the particle-in-cell method implementation, it is shown that the performance achieved by the

LuNA-ICLU is better than that of LuNA system and is comparable to the performance of a manually written MPI program.

2 LuNA-ICLU System

To overcome the problems affecting performance of the LuNA system, the LuNA-ICLU system is developed. As described above, performance problems of LuNA system are basically caused by using universal system algorithms of fragmented program execution. The idea of the LuNA-ICLU system is to apply system algorithms that are able to generate automatically a static MPI program from strongly defined class of fragmented programs. So, the applied program developer does not have to solve the system parallel programming problems such as developing of dynamic load balancing algorithms.

To generate a static MPI program from a given fragmented program it is necessary to analyze information dependencies between CFs described in the input fragmented program. Expressions of the LuNA language use CFs and DFs, including the indexed ones, which are parts of fragmented arrays. Index expressions can be complex and difficult to analyze. To overcome this problem a limited class of input fragmented programs is defined. In addition, the LuNA language was extended by certain high-level statements, which are described below.

In the current implementation of the LuNA-ICLU compiler the class of supported algorithms is the following. The fragmented program can contain 1D or 2D fragmented data arrays (arrays of DFs) and iteration processes described via "while" operator. DF values on current iteration are computed from a set of DF values from one or more previous iterations. The dimensions of DF arrays are strictly separated into temporal, over which iterations go, and spatial. Within iteration each element of DF array may be computed by CF from the elements of DF arrays with corresponding spatial dimension indices being the same. For example, DF A[i] can be computed from B[i], but not from B[i+1] or B[i*2]. The sizes of the corresponding spatial dimensions of different arrays must also coincide. The other types of dependencies should be supported in the language and compiler by special operators (see below). Such a class of algorithms is simple enough for compiler to analyze and contains solutions for many applied problems. In this paper a fragmented program for the PIC method solver is described. In future the class of supported input programs can be extended by implementing certain analyzing and code generating modules for compiler.

3 LuNA Language Extension

In order to overcome the problems of the fragmented program static analysis, the LuNA language has been extended by new syntactic constructions.

- The "DFArray" statement defines an array of DFs (its structure and sizes) that should be distributed among the nodes of multicomputer.

- Among the dimensions of the DF arrays, “spatial” and “temporal” dimensions are clearly distinguished. A “spatial” dimension is denoted by the symbols “[” and “]” and defines a set of DFs that correspond to the same iteration of the iterative process. A “temporal” dimension is denoted by the symbols “(” and “)” and defines different iterations of the iterative process.
- Data dependencies between DF array elements on different iterations of “while” loop are specified explicitly in a loop header using expressions such as: `<A(i-1), A(i) --> A(i+1)>`.
- The “borders_exchange” and “reduce” operators define frequently met templates of structured CFs over arrays of DFs in order to simplify the process of information dependencies analysis and to apply a special optimized implementation in a target program.
- The “dynamic” statement marks a set of CFs in the iteration body that may cause a load disbalance.

4 System Algorithms in LuNA-ICLU System

4.1 Control-Building Algorithm

Since the idea of the LuNA-ICLU system is to generate a static MPI-C++ program from a fragmented program written in LuNA-ICLU language, there is a necessity to design an algorithm that take a fragmented program as input and convert it to a fragmented program with defined control, i.e. it should define a partial order relation on a set of CFs.

In this paper, the bulk synchronous parallel (BSP) model for the target MPI program was considered. Thus, a sequence of CF calls interleaved with communication stages should be built for each MPI process. CFs with spatially distributed indices are distributed among MPI processes according to a distribution function (see below), while the calls to the other CFs are duplicated in each MPI process. The control-building algorithm follows the requirement that each CF must have all its input DF values computed and stored in the memory of the corresponding MPI process before it can be executed. The communication stages of the target MPI program comprise operations such as DF boundaries exchange, reductions, load balancing, etc.

4.2 Arrays Distribution Algorithm

To generate an MPI program from the fragmented program it is required to generate a distribution of DFs by MPI processes. In the current implementation only DFs that are elements of DF arrays are distributed. All other DFs are duplicated in all MPI processes. Indexed CFs are distributed in accordance with indexed DFs they produce.

In the target MPI program the distribution is defined by a mapping function that maps spatial coordinates of array elements to MPI processes. Compiler should generate this function and emit it to the target MPI program. The requirement to the distribution generation algorithm is that it should provide the distribution of DFs that is as close as possible to a uniform. A naive algorithm is applied in the LuNA-ICLU compiler. It considers DFs to be of the same weight, so each DF array dimension is divided by a corresponding size of the Cartesian MPI communicator.

4.3 Dynamic Load Balancing Algorithm

A "dynamic" statement is used by LuNA program developer to tell the compiler that a given subset of CFs can cause a load disbalance on multicomputer nodes at runtime. Compiler should generate the call of load balancing algorithm implementation from LuNA-ICLU runtime library or inline the implementation of some dynamic load balancing algorithm to the output program in order to execute such kind of CFs efficiently.

In the LuNA-ICLU system the dynamic load balancing algorithm is implemented in a runtime library and the compiler inserts calls of corresponding implementation to output program. The load balancing algorithm itself meets the following requirements.

- The algorithm must overcome the load disbalance by changing the mapping function (see Sect. 4.2). At load balancing stage, DFs from overloaded multicomputer nodes are transferred to underloaded ones.
- The algorithm should be parameterized. This requirement is caused by a necessity to tune the algorithm for different applied algorithms and supercomputers. Examples of such parameters are unbalance threshold and frequency of load measurement. In the future versions of the system the execution profile analysis is going to be applied in order to tune the parameters automatically.

In the current implementation a dynamic diffusion load balancing algorithm is applied. In the description below we consider two DFs as neighbors if both DFs are the components of the same DF array and one of their corresponding indices differs by one. We also consider two processes as neighbors if these processes store neighboring DFs. Each process of the target MPI program stores a set of DFs' values that are available locally and a list of each DF's neighbors. The algorithm itself is the following:

1. Each process checks if there is a necessity to call the load balancer (the current iteration number of the iteration process is used).
2. Each process exchanges its current load value (which is basically a measured time spent on execution of CFs specified by the "dynamic" block) with all its neighboring processes.
3. Each process is searching for a neighbor with a maximum load difference compared to itself.
4. If the maximum loads difference is greater than the minimum disbalance threshold (which is basically a parameter of the algorithm), then the process calculates the number of DFs to be sent to the found neighboring process and selects certain DFs.

5. Each process exchanges the information about selected DFs and their neighbors with all neighboring processes.
6. Each process exchanges the values of selected DFs with neighboring processes.
7. Each process updates information about stored DFs and their neighbors.

The considered algorithm has several disadvantages. For example, restriction to local communications may cause a load gradient within a load threshold between neighboring processes, but with a large disbalance between distant processes. In addition, the number of neighboring processes may increase to a large value, which will increase the overhead of load balancing. However, as can be seen from the next section, the algorithm can be applied to resolve the load disbalance appeared when executing fragmented programs.

5 Performance Evaluation

To evaluate the performance of the program obtained by the LUNA-ICLU compiler a test problem of gravitating dust cloud simulation is considered [14]. The simulation algorithm is based on the particle-in-cell method [15]. Parameters of the simulation used in all test runs were the following: mesh size $160 \times 160 \times 100$, number of particles 500 000 000, number of time steps 800. Initial particles distribution was a ball with uniform density located in the center of the simulation domain. The domain decomposition in two directions into 16×16 fragments was applied, so that only several fragments in the center contain particles. Since the main computational load is associated with particles, such problem statement leads to a load imbalance.

Three implementations of the algorithm were developed, using MPI, LuNA and LuNA-ICLU. Moreover, two versions of the programs generated by the LuNA-ICLU compiler were compared: with load balancing and without it. The parameters of the load balancer were the following: the balancing module was invoked every fifth time step, the minimum disbalance threshold was set to 10%. All tests were run using 16 nodes of the MVS-10P Tornado cluster (16 cores per cluster node, 256 cores in total) [16]. The hand-coded MPI program and the MPI program generated by LUNA-ICLU compiler were run using one MPI process per core, whereas the LuNA program was run with one process per node and 16 working threads per process.

Figure 1 shows execution times obtained for different parallel implementations of the considered application algorithm. LuNA-ICLU implementation without load balancing outperforms the LuNA implementation by 10%, whereas with load balancing enabled the execution time decrease is 33%. Hand-written and manually optimizes MPI program even without load balancing outperforms all the other implementations, presumably due to more efficient memory management.

Figure 2 shows the dynamics of time spent by all cores at each time step on useful calculations compared to the time spent on communication operations, including waiting, when running LuNA-ICLU implementations. Without the load balancing enabled, calculations took up only 20% of the total time, whereas load balancing increased this fraction to 45% (60% in the steady state at the end of the simulation).

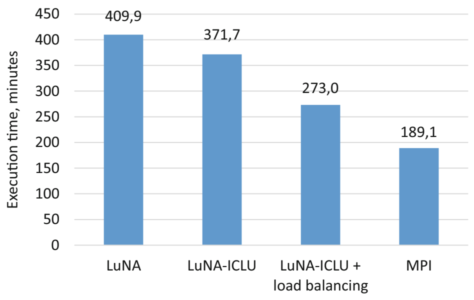

Fig. 1. Execution time for different parallel implementations

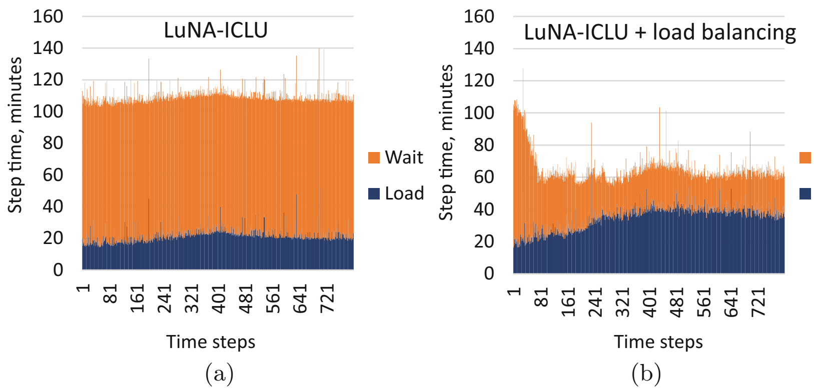

Fig. 2. Dynamics of time spent by all cores at each time step on calculations (Load) and communication operations, including waiting (Wait): LuNA-ICLU implementation without load balancing (a), LuNA-ICLU implementation with load balancing (b)

6 Conclusion

The paper takes a step towards improving the performance of fragmented programs. The problems of the previously developed LuNA system were considered and the prototype of LuNA-ICLU compiler was presented. The results of the performance evaluation are given. It was demonstrated that the performance of LuNA-ICLU system obtained on a PIC method implementation is better than that of the LuNA system and close to the performance of the manually written MPI program. The dynamic load balancing algorithm in the automatically generated MPI program provides a speedup of 1.3 times on the considered problem. The developed fragmented program compiler can be used to automatically generate efficient parallel programs from fragmented programs. In the future, compiler modules can be improved, giving the compiler the ability to support a more complex class of fragmented programs and generate more efficient MPI programs.

References

1. Kessler, C., Keller, J.: Models for parallel computing: review and perspectives. PARS Mitt. **24**, 13–29 (2007)
2. Sterling, T., Anderson, M., Brodowicz, M.: A survey: runtime software systems for high performance computing. Supercomput. Front. Innovations: Int. J. **4**(1), 48–68 (2017). https://doi.org/10.14529/jsfi170103
3. Thoman, P., Dichev, K., Heller, T., et al.: A taxonomy of task-based parallel programming technologies for high-performance computing. J. Supercomput. **74**(4), 1422–1434 (2018). https://doi.org/10.1007/s11227-018-2238-4
4. Legion Programming System. http://legion.stanford.edu. Accessed 23 May 2019
5. HPX - High Performance ParalleX. http://stellar-group.org/libraries/hpx. Accessed 23 May 2019
6. Mattson, T.G., et al.: The open community runtime: a runtime system for extreme scale computing. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2016). https://doi.org/10.1109/HPEC.2016.7761580
7. Charm++. http://charm.cs.illinois.edu/research/charm. Accessed 23 May 2019
8. Regent: a Language for Implicit Dataflow Parallelism. http://regent-lang.org. Accessed 23 May 2019
9. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: a generic distributed DAG engine for high performance computing. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Ph.d Forum, Shanghai, pp. 1151–1158 (2011). https://doi.org/10.1109/IPDPS.2011.281
10. PaRSEC - Parallel Runtime Scheduling and Execution Controller. http://icl.utk.edu/parsec. Accessed 23 May 2019
11. Malyshkin, V.E., Perepelkin, V.A.: LuNA fragmented programming system, main functions and peculiarities of run-time subsystem. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 53–61. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23178-0_5
12. Akhmed-Zaki, D., Lebedev, D., Perepelkin, V.: Implementation of a three dimensional three-phase fluid flow ("Oil-Water-Gas") numerical model in LuNA fragmented programming system. J. Supercomput. **73**(2), 624–630 (2017). https://doi.org/10.1007/s11227-016-1780-1
13. Alias, N., Kireev, S.: Fragmentation of IADE method using LuNA system. In: Malyshkin, V. (ed.) PaCT 2017. LNCS, vol. 10421, pp. 85–93. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62932-2_7
14. Kireev, S.: A parallel 3D code for simulation of self-gravitating gas-dust systems. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 406–413. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03275-2_40
15. Hockney, R.W., Eastwood, J.W.: Computer Simulation Using Particles. IOP Publishing, Bristol (1988)
16. MVS-10P cluster, JSCC RAS. http://www.jscc.ru. Accessed 23 May 2019