LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem

V.E.Malyshkin and V.A.Perepelkin

Institute of Computational Mathematics and Mathematical Geophysics
Russian Academy of Sciences
Novosibirsk

{malysh, perepelkin}@ssd.sscc.ru
http://ssd.sscc.ru

Abstract. The peculiarities of the LuNA run-time subsystem implementation are considered. LuNA is the language and system of fragmented programming. The peculiarities are conditioned by the properties of numerical algorithms, to implementation and execution of which the LuNA is mainly oriented.

Keywords: run-time, parallel and distributed computing, parallel programming language and system, model of parallel program.

1 Introduction

The idea of data and algorithms fragmentation has been exploited in programming at least since the early 1970-s [1–8]. This approach to computation organization with the use of a run-time subsystem is used if the solutions on how to execute a program or its parts, how to distribute the resources should be done dynamically. Different modifications of this approach were embodied in programming systems [2–5]. Many programming systems use the run-time subsystems for the organization of computation [5–13]. In [2] instead of commonly used run-time system for program execution, a special hardware and operating system were developed. Our project of the LuNA fragmented programming system is oriented to the creation of a parallel numerical subroutine library.

2 Introductory Definitions

A general model of a program in the above mentioned systems can be described as computational model [3].

2.1 General Model Definition

Given:

- The finite set **X**={*x, y, ..., z*} of variables for representation of different computed values;
- The finite set **F**={*a, b, ..., c*} of functional symbols (operations, Fig. 1.a), $m\geq 0$ is the number of input variables, $n\geq 0$ is the number of output variables;
- in(a)=(x_1, ...,x_m) is a set of input variables, out(a)=(y_1, ...,y_n) is a set of output variables (Fig. 1), if $i\neq j \rightarrow y_i \neq y_j.\& \ x_i \neq x_j$

Model C=(**X**,**F**) is called simple computational model (SCM). Operation $a\in$**F** describes the possibility to compute the variables out(a) from the variables in(a), for example, with the use of a procedure. The model can be graphically depicted (fig. 1.b, 1.c)

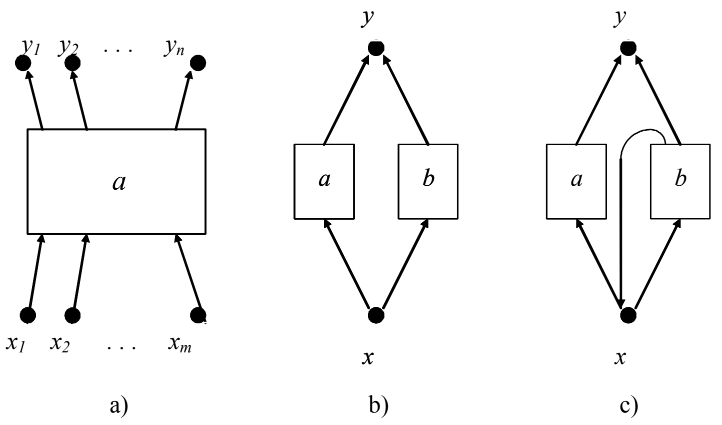

Fig. 1. Examples of operations, variables and models

Let $V\subseteq$**X**, $F\subseteq$**F** be given. A set of functional terms *T(V,F)* is defined as follows:

1. If $x\in V$, then x is a term t, $t\in T(V,F)$; in(t)={x}; out(t)={x}.
2. Let {t^1, ..., t^s} $\subseteq T(V,F)$ and $a\in F$, in(a)=(x_1,...,x_s) be given. The term t=a(t^1,...,t^s) is included into $T(V,F)$ if $\forall i(x_i\in$out(t^i)), in(t)=$\bigcup_{i=1}^{s}$ in(t^i), out(t)=out(a). Here t=a(t^1,...,t^s) denotes that t is the term a(t^1,...,t^s).

A term is depicted as a tree that contains both operations and variables of the term, see Fig. 2.a.

We say that a term t computes a variable y, if $y\in$out(t). The set of terms *T(V,F)* defines all the variables of the SCM, that can be computed from V variables. A set of terms T_V^W ={$t\in T(V,F)$| out(t)$\cap W\neq\varnothing$} computes all the variables from W that can be computed from V variables.

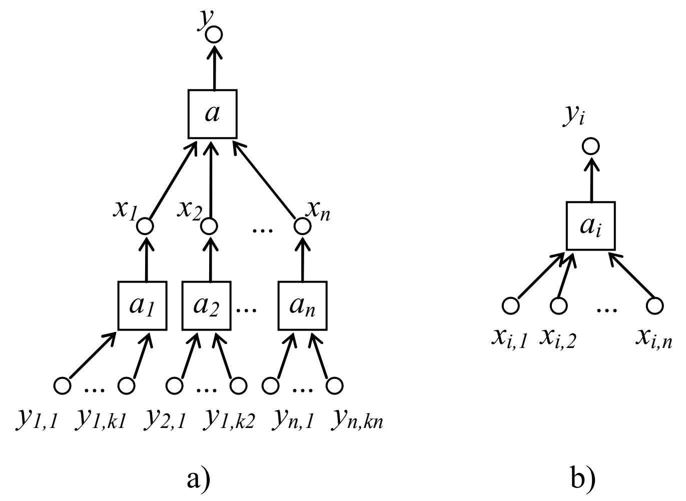

Fig. 2. Depicted terms

Any such subset $R \subseteq T_V^W$ that $\forall x \in W \exists t \in R(x \in \text{out}(t))$ is called (V,W)-plan and defines an algorithm computing the variables W from the variables V. Here V and W denote the sets of input and output variables of the algorithm respectively. Everywhere further a set of recursively countable functional terms is considered as a representation of an algorithm.

2.2 Interpretation

Let $V \subseteq \mathbf{X}$ be given. Interpretation $\boldsymbol{I}$ in the domain D is the function that assigns to:

- every variable $x \in V$ an entry $d_x = \boldsymbol{I}(x) \in D$, d_x is a value of the variable x in the interpretation $\boldsymbol{I}$,
- to every operation $a \in \mathbf{F}$, in$(a)=\{x_1, x_2, ..., x_m\}$, out$(a)=\{y_1, y_2, ..., y_n\}$, a computable function f_a: $D^m \to D^n$,
- to every term $t=a(t_1,t_2,...,t_m)$ a superposition of the functions accordingly to the rule $\boldsymbol{I}(a(t_1,t_2,...,t_m))=f_a(\boldsymbol{I}(t_1),\boldsymbol{I}(t_2),...,\boldsymbol{I}(t_m))$.

If $t=a(t_1,t_2,...,t_m)$ is an arbitrary term, in$(a)=\{x_1, x_2, ..., x_m\}$, out$(a)=\{y_1, y_2, ..., y_n\}$, then $\boldsymbol{I}(\text{out}(a))=val(t)=(d_1,d_2,...,d_n)=f_a(valx_1(t_1),valx_2(t_2),...,valx_n(t_n))$.

Further it is assumed that for every function $f_a=\boldsymbol{I}(a)$ there exists a module (procedure) mod_a that can be used in a program to compute the function f_a.

2.3 Correct Interpretation

If there exist two different terms t_1 and t_2, $y \in \text{out}(t1) \cap \text{out}(t2)$, in$(t1) \cup$in$(t2) \subseteq V$, then $valy(t1)=valy(t2)$ in the interpretation $\boldsymbol{I}$, the interpretation $\boldsymbol{I}$ is called correct interpretation. In the correct interpretation for any variable y, any pair of terms t_1 and t_2, $y \in \text{out}(t_1) \cap \text{out}(t_2)$ yields the same value, $valy(t_1)=valy(t_2)$.

For definition of mass computations this model is extended by the inclusion indexed operations and indexed variables (arrays), fig. 2.b. This technical work can be easily done. Obviously, in this extended model, a mass algorithm is represented by a potentially infinite recursively countable set of functional terms [3].

A program that implements an algorithm, represented by a set of functional terms, can be constructed with the procedure calls to mod_a done in the order that does not contradict to the information dependences between the operations imposed by the terms structures. Usually, a run-time subsystem is used to implement all the calls in the proper order.

3 The LuNA Fragmented Programming System

3.1 Model Modifications

The LuNA fragmented programming system implements the above model. In order to provide the reachability of high performance of a fragmented program execution the above model was essentially transformed [14]:

- data and computation fragmentations were included into the model [15–17]; the subsets of functional terms are aggregated into bundles, forming aggregated variables (data fragments – DF) and aggregated operations (fragments of computations – FC),
- multiple assignment variables were included, whereas every FC is permitted to be executed once only,
- the users's recommendation are introduced into an algorithm description, that are used by the LuNA run-time subsystem for improvement of the fragmented algorithm execution.

Additionally, LuNA applications are restricted by the numerical algorithms. Regular structure of mesh numerical algorithms essentially simplifies the algorithms of the LuNA run-time subsystem implementation. As result, the algorithms of LuNA run-time subsystem provide high performance execution of numerical algorithms and don't guarantee good execution of the algorithms from the another application area. The above modifications allowed essential improvement of the algorithms of the LuNA run-time subsystem implementation providing high performance of the LuNA programs. All the above modifications preconditioned the main features and peculiarities of the LuNA run-time subsystem implementation.

3.2 Scheme of the LuNA Implementation

LuNA program (FP – fragmented program) is constructed in two main stages:

a). an application algorithm fragmentation [15–17] and its representation by the set of fragments of computations (operations in the above model), the set of data fragments and partial order relation ρ in order to define the information dependences between fragments of computations (FC). Representation of the algorithms with the LuNA input language, including user's recommendations.

b). such a representation is already considered as an executable FP. The execution is organized in next 3 steps:

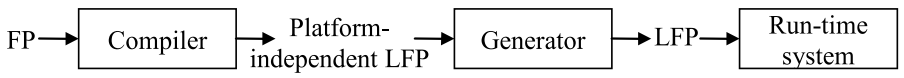

Fig. 3. The 3 steps of FP execution

- **Compilation**. On this step all the decisions (construction of the initial resources allocation and control), that can be done statically, are made. Also the compiler analyses the FP and proposes one or more preliminary schemes of the FP execution and resources assignment. For example, several DFs can be assigned to a one memory slot in order to save the memory; or a number of CFs can be folded into a loop by defining additional direct control. The initial resources allocation is not fully constructed on this stage in order to provide the desirable level of asynchronism in the course of the FP execution. The compiler also checks the FP in order to recognize the syntax errors and to accomplish some other traditional compilation tasks too. The compiler has no information about hardware configuration or input data. The result of the compilation is the platform-independent FP.
- **Generation**. On this step all the decisions whose making depends on the computer system architecture and its configuration are generated. The generator takes the platform-independent FP as input and a description of a certain computer system configuration. The generator defines the parameters of the FP, such as DF size, in order to fit better the hardware. It also selects one of schemes of FP execution (if the compiler has provided more than one), which better fits the given hardware configuration. The result of generation is FP, executable by the run-time subsystem.
- **Execution** is provided by the LuNA run-time subsystem. The set T_V^W of the functional terms is not really constructed, the necessary term is constructed if necessary only. The run-time subsystem provides dynamic properties of the FP execution, such as dynamic workload balancing. In the course of the FP execution the run-time subsystem is capable to change the order of the FC execution and resources allocation schemes in order to optimize the efficiency of FP execution in run-time.

In order to provide the high performance of an FP execution two main problems should be solved by the LuNA run-time subsystem: to choose and to assign for execution a certain FC and to construct the rest of resources allocation.

3.3 LuNA Input Language

As usual [3], the LuNA input language contains the facilities for FC, DF and control ρ description. Also it contains the user's recommendations, which provide the compiler and the run-time subsystem with the additional information on how to improve the execution of the FP. More detailed description of the FP representation and its peculiarities are considered in [14].

Consider an example of DF, FC and ρ definitions, written with the input LuNA language:

Table 1. Example of DF, FC and ρ definitions in LuNA language

Definitions of a FP	**Description**
df x[i] : block(real, M) | i=1..N;	Definition of N DFs x[i], each containing M entities of real type.
cf a[i] : func_a(in: x[i]; out: y[i], z[i]) | i=1..N;	Definition of N FCs a[i] with specified input and output DFs. For implementation of FCs a[i] the func_a procedure is assigned.
a[i] < a[i+1] | i=1..N-1;	A set of pairs <a[i], a[i+1]> is included into ρ.

LuNA run-time subsystem utilizes the following types of recommendations.

Priority. Priority is a real-valued function, defined on the set of FCs. If there is an ability for run-time subsystem to fetch for execution a number different FCs, then the ones with the highest priority are fetched. Definition of the priorities allows controlling the flow of FCs execution in order to optimize performance (see profiling subsection below). Like the rest recommendations, execution of the FP according to the priorities is not mandatory. Depending on different factors of a certain situation, run-time subsystem can accept or not the defined recommendations.

Neighborhood Relation. Binary relation called neighborhood relation is defined on the set of DFs. Two DFs are defined to be neighbor-related if it is recommended to keep them close to each other, for example, in the memory of the same PE. Usually this is done for DFs, which are the input variables of the same FC, and location of them in the same PE leads to reduction of the total communication overhead. Neighborhood relation is used by run-time subsystem in the process of dynamic workload balancing. If some workload has to be transferred from one PE to another, the run-time subsystem tries to minimize the number of neighbor-related DFs in different PEs after workload migration. Neighborhood relation provides the use of regularity of data and computation structures of numerical algorithms for the optimization of their execution.

Execution Template. To optimize the performance of the most time- and resource-consuming parts of the FP an execution template (ET) can be defined. The ET is an oriented graph, ET=<N,E>, where N is the set of ET nodes, and E is the set of ET oriented edges, $E=\{<n_1,n_2>|n_1,n_2\in N\}$. The nodes N are execution units, connected with each other by edges, which transfer values from one node to another. A number of FCs can mapped to ET nodes. Consider an example (fig. 4). FCs a_i are mapped to node A, b_i – to node B and FC c is mapped to node C. Execution of the part of FP, mapped to the ET is organized as follows. When a node has a value on each of the incoming edges, it executes corresponding FC without any additional checks. After execution the output values are promoted via output edges to other nodes. In such a way, ET specifies inflexibly the scheme of FCs execution, which reduces the run-time subsystem overhead of choosing FC for execution.

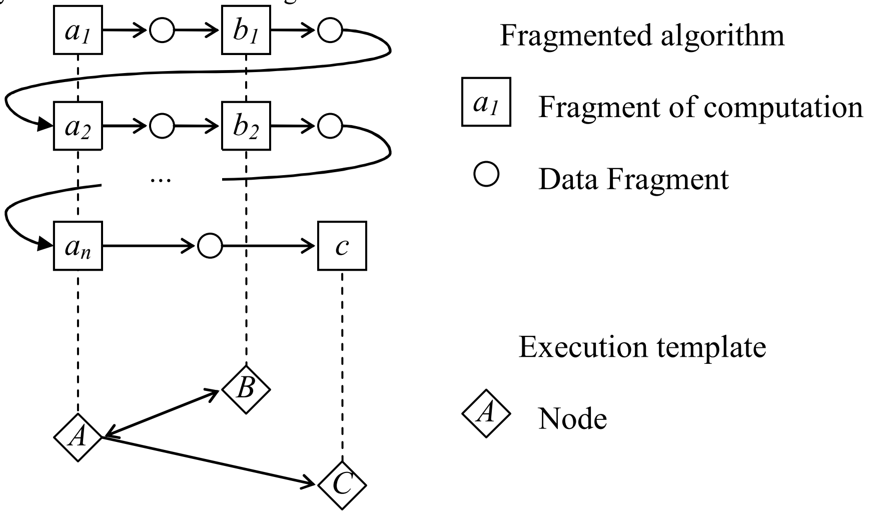

Fig. 4. FP to ET mapping.

Profiling. Profiling is a process of gathering a profile, which is the run-time information about an FP execution. This information includes real order of FCs execution, FCs execution times, PEs workload over time information, etc. The profile is used by run-time subsystem to optimize next executions of the FP. For example, if during the FP execution some PEs were idle because the value of some DF x was not calculated in time, then this information will be extracted from the profile, and during the next execution of the FP the FCs, which provide yielding the value of x will be assigned for execution earlier (if possible). It can be achieved, for example, by increasing the priority of these FCs. In such a way, each next execution of the FP will be done more efficiently (up to some limit, of course), if the FP is run on the same multicomputer and with the similar input data

3.4 Testing.

The performance of the LuNA run-time subsystem was tested on the implementation of a numerical model of self-gravitating stardust cloud using Particle-In-Cell method [18]. The parallel MPI-based implementation of the model was compared to the same implementation in LuNA programming system. The model was implemented as a fragmented program, using approach, described in [15]. The FP was executed by the LuNA run-time subsystem. The testing was performed on a cluster of the Siberian Supercomuter Center [19]. The testing results are shown in fig. 5.

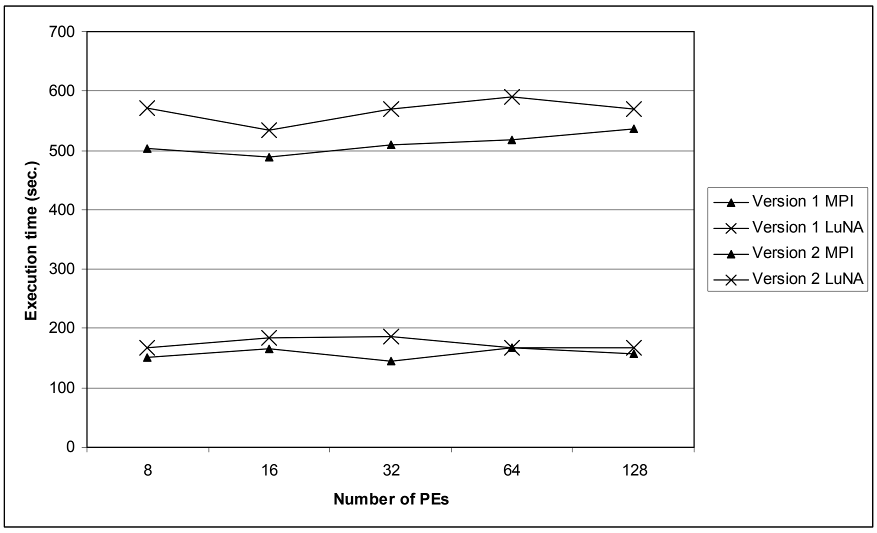

Fig. 5. Test results.

There were two versions of the test, which differ by problem size. In the version 1 mesh size was 100×100×100 and the number of particles was 10^6. In the version 2 mesh size was 200×200×200 and the number of particles was 10^7. In both versions the problem size was increased with the PEs count increase (in the same proportion). In such a way, the execution time was approximate the same for the same version of the test.

As it is seen in the fig. 5 the difference in execution time between MPI and LuNA implementation is minor, which means, that the efficiency of hand-made MPI programs is reachable using LuNA programming system. However, to reach that efficiency, a FP has to be tuned up by the properly defined recommendations.

4 Conclusion

Taking into account the peculiarities of numerical algorithm provided low level of LuNA overhead, high performance of numerical algorithms implementation. Some additional LuNA modification are also planned to be implemented soon.

References

1. Glushkov V.M., Ignatiev M.V., Myasnikov V.A., Torgashev V.A.: Recursive machines and computing technologies. - In Proceedings of the IFIP Congress, North-Holland Publish. Co, Vol.1., pp. 65-70 (1974).
2. Torgashev V.A., Tsarev I.V. Programming facilities for organization of parallel computation in multicomputers of dynamic architecture. - Programmirovanie, No.4, pp. 53-67

(2001) (In Russian) (Sredstva organizatsii parallelnykh vychislenii i programmirovaniya v multiprocessorakh s dynamicheskoi architechturoi).

3. Valkovskii V., Malyshkin V.: Parallel Program Synthesis on the Basis of Computational Models. – Novosibirsk, Nauka (In Russian. Sintez parallel'nykh program i system na vychislitel'nykh modelyakh) (1988)
4. Cell Superscalar, http://www.bsc.es/cellsuperscalar
5. Charm++, http://charm.cs.uiuc.edu
6. Shu W., Kale L.V.: Chare Kernel – a Runtime Support System for Parallel Computations. – Journal of Parallel and Distributed Computing, Volume 11, Issue 3, pp 198-211 (1991)
7. Kalgin K.V., Malyskin V.E., Nechaev S.P., Tschukin G.A.: Runtime System for Parallel Execution of Fragmented Subroutines. – In Proceedings of the 9th International conference on Parallel Computing Technologies (PaCT-2007), LNCS, Vol. 4671, Springer, pp 544-552 (2007)
8. Blumofe R.D., Joerg C.F., Kuszmaul B.C., Leiserson C.E., Randall K.H., Zhou Y.: Cilk: An Efficient Multithreaded Runtime System. – ACM SIGPLAN Notices, Volume 30, Issue 8, pp 207-216 (1995)
9. Foster I., Kesselman C., Tuecke S.: Nexus: Runtime Support for Task-Parallel Programming Languages. - Cluster Computing, Issue 1(1), pp 95-107 (1998)
10. Chien A.A., Karamcheti V., Plevyak J.: The Concert System – Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs. – UIUC DCS Tech Report R-93-1815 (1993)
11. Grimshaw A.S., Weissman J.B., Strayer W.T.: Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. – ACM Transactions on Computer Systems (TOCS), Volume 14, Issue 2, pp 139-170 (1996)
12. Benson G.D., Olsson R.A.: A Portable Run-Time System for the SR Concurrent Programming Language. – In Proceedings of the Workshop on Run-Time Systems for Parallel Programming (RTSPP) (1997)
13. Jack J. Dongarra, Danny C. Sorensen and Sven J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. // Journal of Computational and Applied Mathematics, Vol. 27, Issues 1-2, September 1989, Pages 215-227. Special Issue on Parallel Algorithms for Numerical Linear Algebra
14. V.Malyshkin and V.Perepelkin. Optimization of Parallel Execution of Numerical Programs in LuNA Fragmented Programming System. – In the Proceedings of the second Russia-Taiwan symposium on Method and tools of Parallel Programming Multicomputers (MTPP). Springer Verlag, LNCS series, Vol. 6083, pp. 1-10, 2010.
15. Kraeva M.A., Malyshkin V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. - In the Int. Journal on Future Generation Computer Systems, Elsevier Science. Vol. 17, No. 6, pp 755-765 (2001)
16. Handbook of Research on Scalable Computing Technologies. // IGI Global, USA, 2010, 1021 pp. ISBN 978-1-60566-661-7
17. V.Malyshkin, S.Sorokin, K.Chauk. Fragmentation of numerical algorithms for the Parallel Subroutine Library. – Springer Verlag, 2009, LNCS series, Vol.5698, pp. 331-343.
18. Kireev S.E. Parallel Implementation of the Particle-In-Cell Method for Modeling of Problems of Gravitation Cosmo-Dynamics (In Russian: Parallelnaya Realizaciya Metoda Chastits v Yacheykah Dlya Modelirovaniya Zadach Gravitacionnoy Kosmodinamiki) // Avtometriya, No. 3, 2006. pp. 32–39.
19. Siberian Supercomputer Center, http://www.sscc.ru