Journal of Supercomputing manuscript No.
(will be inserted by the editor)

The PIC Implementation in LuNA System of Fragmented Programming

V.E.Malyshkin · V.A.Perepelkin

Received: date / Accepted: date

Abstract The main features of the LuNA system of fragmented programming, aimed at parallel implementation of the large-scale numerical models on the mesh, are considered. The complex application of the Particle-In-Cell method (PIC) to a large-scale 3D dust-cloud model, developed in LuNA, demonstrates its advantages for providing such dynamic properties of application programs as portability, dynamic load balancing and tunability to all the available resources.

Keywords particle-in-cell method parallel implementation · dynamic load balancing · fragmented programming technology · dust cloud simulation

1 Introduction

The LuNA programming system is aimed at the provision of the automatic generation of parallel programs for the large scale numerical modeling. The LuNA program is assembled out of data and computation fragments, i.e., ready-made modules. The idea of the data and algorithms fragmentation has been exploited in programming over a period of years [1–6] Different modifications of this approach were embodied in programming systems [2–5]. Most of programming systems used run-time systems for the organization of computation [7–13].

Charm++ [4] program is assembled out of modules (Chares), but algorithm representation is done with partial resources allocation. This substantially reduces the portability and tunability of an application Charm++'s program.

Victor Malyshkin · Vladislav Perepelkin
Institute of Computational Mathematics and Mathematical Geophysics,
Russian Academy of Sciences
Tel.: +7(383)330-89-84
E-mail: {malysh, perepelkin}@ssd.sscc.ru

Also, Charm++ does not have the means for data allocation and control definition.

In PaRSEC [5,6], data are assigned to certain nodes that leads to a decrease of portability. Also, a change of operation interpretation is restricted. Certainly, this is an inherent property of the D-PLASMA library.

In [2], instead of a commonly used run-time system, a specialized hardware and operating system for organization of the programs execution were developed.

2 The LuNA approach to a program construction

The LuNA fragmented programming system [14] is aimed at the development of the application software, implementing large-scale numerical models. The LuNA program is assembled out of data and computation fragments, i.e., ready-made modules. In order to attain a high performance of application program developed in the LuNA, the LuNA system software exploits such properties of numerical algorithms as regularity of computations, data and communications, the existence of a mesh, the neighborhood relation both on data and on computations processing these data. In this paper, advantages of the LuNA approach are demonstrated on the example of programming of the Particle-In-Cell method (PIC) applied to the astrophysics problem of dust cloud simulation [15].

3 Model of computation

The LuNA model of computation is based on the method of parallel program synthesis on the basis of computational models [1]. The method proposes a set of theoretical representations of algorithms and programs.

3.1 Formal model

Given:

- The finite set $\boldsymbol{X} = \{x, y, ..., z\}$ of variables for representation of different computed values;
- The finite set $\boldsymbol{F} = \{a, b, ..., c\}$ 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 $C = (\boldsymbol{X}, \boldsymbol{F})$ is called *simple computational model* (CM) [1]. Operation $a \in \boldsymbol{F}$ describes the possibility to compute the variables $out(a)$ from the variables $in(a)$, for example, with the use of a certain procedure.

Let us consider the notions of a functional term and algorithm in this model. Let $V \subseteq \boldsymbol{X}$, $F \subseteq \boldsymbol{F}$ 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 $\forall i(x_i \in out(t^i))$, $in(t) = \cup_{i=1}^{s} in(t^i)$, $out(t) = out(a)$. Here $t = a(t^1, ..., t^s)$ denotes that t is the term $a(t^1, ..., t^s)$. A term can be depicted as a tree that contains both operations and variables of the term [1]

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

Any subset $R \subseteq T_V^W$ such that $\forall x \in W \exists t \in R(x \in out(T))$ 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 functional terms is considered as the representation of an algorithm.

3.2 Interpretation

Let $V \subseteq \boldsymbol{X}$ be given. *Interpretation* $\boldsymbol{I}$ in the domain D is a 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 $a \in \boldsymbol{F}$, $in(a) = \{x_1, x_2, ..., x_m\}$, $out(a) = \{y_1, y_2, ..., y_n\}$, 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 is assigned in accord with 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 $\boldsymbol{I}(out(a)) = val(t) = (d_1, d_2, ..., d_n) = f_a(val_{x_1}(t_1), val_{x_2}(t_2), ..., val_{x_n}(t_n))$.

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

For definition of mass computations this model should be extended by inclusion of indexed operations and indexed variables (arrays) [1]. This technical work can be easily done.

A program that implements an algorithm, represented by a set of functional terms, can be constructed with the procedure calls to all the mod_a done in the order not contradicting to the information dependences between the functions, imposed by the terms structure.

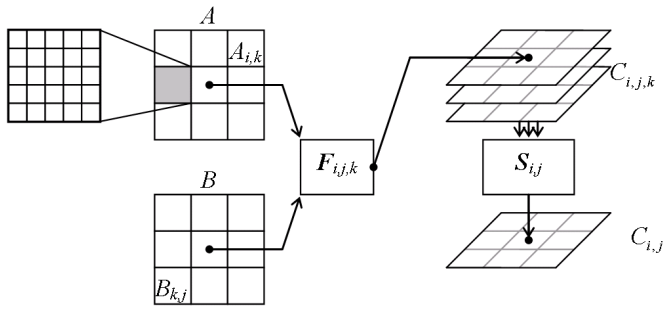

Fig. 1 Fragmented algorithm (the set of terms) of matrices multiplication

4 The LuNA language

The basic algorithm representation in the LuNA is a recursively countable set of functional terms. Omitting the details, let us consider a simple example of the two square $N \times N$ matrices A and B multiplication, $C = A \times B$ [18]. First, all the matrices are *fragmented* and represented as square $K \times K$ matrices of the square $M \times M$ sub-matrices $A_{i,k}, B_{k,j}, C_{i,j}$ (Fig. 1). The sub-matrices $A_{i,k}, B_{k,j}, C_{i,j}$ are called *data fragments* (DF), $N = K \times M$.

The DFs $C_{i,j,k}$ are intermediate DF. The *fragments of computations* (CFs) $F_{i,j,k}$ and $S_{i,j}$ define the sub-matrices multiplication $A_{i,k} \times B_{k,j} = C_{i,j,k}$ and the summation $C_{i,j} = \sum_{k=1}^{K} C_{i,j,k}$, respectively. The neighborhood relation ρ is extracted by the compiler from the information dependences between CFs. The set of functional terms is not constructed. A certain term is constructed, if necessary, by the derivation algorithm [1].

The LuNA fragmented program looks like:

```
sub mul_matrix(name A, name B, name C) {
  df A, B, C, Ctmp;
  for i=0..N-1
    for j=0..N-1 {
      for k=0..N-1
        cf f[i][j][k] mult_mat(A[i][k], B[k][j],
          Ctmp[i][j][k]);
      cf s[i][j] sum_mat(Ctmp, i, j, C[i][j]);
    }
}
```

5 The PIC fragmentation

In [19], the PIC application to modeling an energy exchange in plasma cloud and the PIC algorithms fragmentation are described. The PIC model of plasma contains the two types of data: 3D mesh (space of modeling - SM), and particles. Any particle belongs to a certain cell (Fig. 2). In the course of simulation,

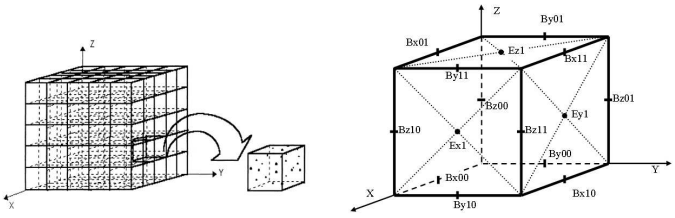

Fig. 2 3-D mesh (SM) and the mesh variables (electrical E and magnetic B fields) distributed on a cell

particles are moving under the influence of electromagnetic fields, discretized upon a 3D mesh, and migrate from one cell to another. The particle processing consists in calculating its new coordinates and its influence on the values of electromagnetic fields.

Data fragmentation is done by dividing a mesh into rectangular blocks: layers or columns or parallelepipeds of cells. The blocks are processed independently, in particular, in parallel. The block processing consists in the processing of all the particles belonging the block and processing all the mesh variables (discretized fields). The volume of computations (the time of processing) of a block depends on the number of particles, located in the block. For processing, initially the blocks are distributed among the processor elements (PE) of a multicomputer in such a way that an equal workload of the PEs is provided.

Because the particles are permanently migrating, even initially an equal workload of PEs is being disbalanced. The automatic dynamic balancing of the workload is the main problem of the PIC parallel implementation.

If a gravitational field is used, the PIC algorithms fragmentation is done in the same way.

6 The PIC programming in the LuNA language

The PIC based astrophysical model of the dust cloud simulation [15] was programmed with the LuNA, tested and the quality of its execution was analyzed. Movement of the dust particles in the gravitational field was simulated. The tests demonstrated that the LuNA overhead is not very high, dynamic properties of the simulating program are provided automatically, programming of the model did not demand having good experience in parallel programming.

As usual, with the LuNA this PIC application is programmed as sets of DFs and CFs. DFs are described here as parallelepipeds of cells. The number of cells, constituting a parallelepiped, can be changed before program execution. Certainly, sequential programs, processing the mesh variables and particles, should be developed by the user.

On a set of DFs the neighborhood relation is defined. All the parallelepipeds adjacent to a certain DF are defined as its neighbors (generally, 26 neighbors).

In the course of a fragmented program execution, the dynamic load balancing of a workload is provided by a special module. The module implements a diffusive load balancing algorithm equalizing the processor elements (PE) workload by means of DFs migration and keeping the neighborhood relation, i.e. the neighboring DFs after balancing are located in neighboring PEs or so. This module is also used in order to construct the initial workload of the PEs. The DFs of a fragmented program are input into a certain PE and then they are dynamically balanced keeping the neighborhood relation while the equal workload of all the PEs is attained.

7 The LuNA testing and analysis

In order to demonstrate the LuNA ability to efficiently execute a fragmented program, the PIC application to modeling of the astrophysics problem of the dust cloud simulation [15] was programmed with the LuNA and program execution was tested.

7.1 General conditions for all the tests.

The space of modeling (SM) is the 3-D cube $64 \times 64 \times 64$ of cells (Fig. 2). Near the central part of SM a dust disc is located. Its diameter is four times less than the length of the cube edge. The other cells contain few particles for background processing. The dust particles are revolving around the center of mass, their velocities being different. Initially uneven particles distribution inside the disc should be changing in the course of simulation equalizing the PEs workload.

Initially equal number of DFs (parallelepipeds of cells $4 \times 4 \times 4$) is assigned for execution into every PE. The number of DFs is equal to $16 \times 16 \times 16$, the number of particles being equal to 10^7 and more. In consequence to initially uneven particles distributions, the PEs are loaded not equally, the workload of PEs is not balanced. The PEs workload imbalance should be equalized in the course of simulation during several iterations. For the correctness of testing, the results of simulation were compared to the results of the hand-made MPI program [15].

Test 1. Dynamic load balancing. The objective of the test is to demonstrate how efficient the load balancing algorithm is. The test was carried out with balancing and without balancing. The workload of PEs was measured every 10 seconds. Balancing should substantially reduce the time of simulation.

Version 1. Program execution without balancing (Fig. 3). In Fig. 3, one can see the axis X shows the time of execution, the axis Y shows PEs number, the axis Z shows the PEs workload on a percentage basis. The A-bloks show the time of particles processing, B the time of Poisson equation solution, C idle PEs. There is a disbalance of the PEs workload, and this disbalance is kept up

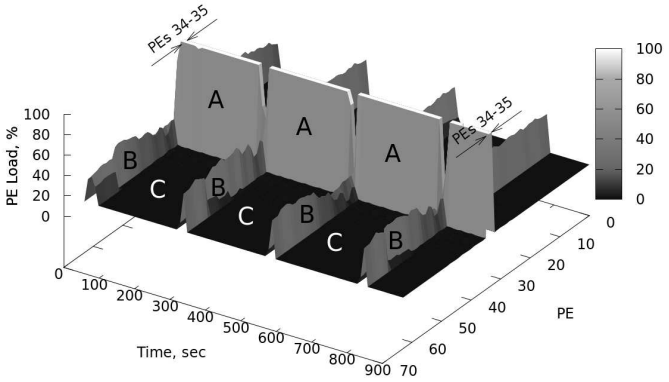

Fig. 3 PEs workload without balancing

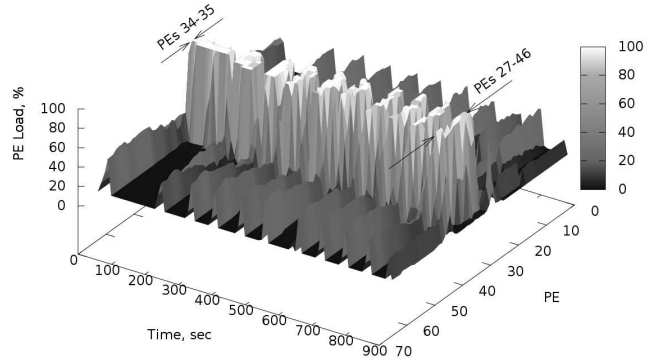

Fig. 4 PEs workload with balancing

to the completion of simulation. The efficiency of the program execution is low, because only the 34^{th} and 35^{th} PEs are working intensively, the others being practically idle (a PEs workload is in proportion to the number of particles, located in the PE). The total time of the program execution is equal to 2320 sec.

Version 2. Program execution with balancing (Fig. 4). The same test is run, balancing is done. By the 800^{th} second from the beginning of the program execution, the PEs from the 46^{th} to the 27^{th} were involved in intensive computation and had already a big enough workload. The time of one iteration execution was also decreased, because the number of particles inside a PE was decreased . The total time of the program execution is equal to 860 sec., i.e., it was decreased almost by the factor of three.

Test 2. The estimation of the LuNA run-time system overhead. The number of DFs was fixed, therefore, the overhead was also fixed. The number of particles

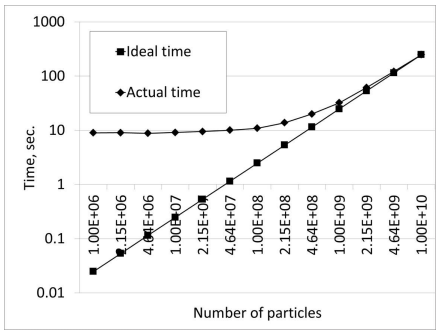

Fig. 5 Overhead of the LuNA run-time system

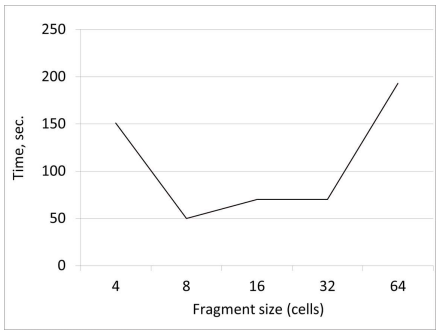

Fig. 6 Time of program execution depending on DFs size

varied from 10^6 to 10^9, the time of simulation also varied proportionally. The parameters of the model were set in such a way, that the time of simulation be close to overhead plus the time of particles processing. From these data, the LuNA overhead was calculated. In Fig. 5 the line shows the time of simulation with zero overhead. The curve shows the results of testing. It is clearly seen that the overhead is close to zero if a model contains 10^8 particles or more. This is not a big model for the PIC applications.

Test 3. Optimization of the execution time of a fragmented program. If only one DF can be located inside a PE, not more, than the LuNA run-time system does practically nothing and its overhead is close to zero. What overhead can be considered as acceptable? What size of DFs can provide an acceptable overhead? The test should answer these questions.

The size of a problem is fixed. The size of DFs varies. The greater is a DF size the less overhead is. The program was executed with different sizes of DFs and the time of execution was measured. The size of every DF is measured as the number of cells (a cube of cells). Fig. 6 shows the time of the program execution depending on a DFs size, the size of a DF is measured by the number of cells along the axis, for example, the Fragment size equal to 4 in Fig. 6 denotes a 3-D cube $4 \times 4 \times 4$ of cells.

As far as the DFs size is increasing, the time of execution is first decreasing because the number of DFs is being less and the overhead is decreasing. Later, the time of execution begins to grow because there are too few DFs and sometimes there is no workload for all the PEs, part of them being idle.

8 Conclusion

The LuNA fragmented programming system automatically provides all the necessary dynamic properties of an application program such as dynamic load balancing, tunability, portability, etc. The LuNA provides a high performance

of an application program and a high level programming of a numerical application. The algorithms of the LuNA system implementation aimed at the parallel implementation of large-scale numerical models on peta- and exa-flops multicomputer. All these properties make the use of the LuNA highly suitable for the large-scale numerical models implementation and their simulation on the heterogeneous multicomputers, supercomputer GRID and cloud [20].

9 Acknowledgments

The reported study was supported by Russian Foundation for Basic Research (RFBR), research projects No. 14-07-00381 A and 14-01-31328 mol_a.

References

1. V.A.Valkovskii, V.E.Malyshkin (1988) Parallel Program Synthesis on the Basis of Computational Models. Nauka, Novosibirsk
2. Torgashev VA, Tsarev IV (2001) Programming facilities for organization of parallel computation in multicomputers of dynamic architecture. Programmirovanie, No.4, pp. 5367 (In Russian).
3. Cell Superscalar, http://www.bsc.es/cellsuperscalar. Accessed 25 December 2013
4. Charm++, http://charm.cs.uiuc.edu. 25 December 2013
5. The Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) project http://icl.cs.utk.edu/plasma. Accessed 25 November 2013
6. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Dongarra, J. "PaRSEC: Exploiting. Heterogeneity to Enhance Scalability," IEEE Computing in Science and Engineering, Vol. 15, No. 6, 36-45, November, 2013.
7. Shu W, Kale LV (1991) Chare Kernel a Runtime Support System for Parallel Computations. J Parallel Distrib Comput, Vol. 11, Issue 3, pp. 198211
8. Kalgin KV, Malyskin VE, Nechaev SP, Tschukin GA (2007) Runtime System for Parallel Execution of Fragmented Subroutines. 9th Int Conf Parallel Comput Technol, Springer Verlag, LNCS, Vol. 4671, pp. 544552
9. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y (1995) Cilk: An Efficient Multithreaded Runtime System. ACM SIGPLAN Not, Vol. 30, Issue 8, pp. 207216
10. Foster I, Kesselman C, Tuecke S (1998) Nexus: Runtime Support for Task-Parallel Programming Languages. Cluster Computing, Issue 1(1), pp. 95107
11. Chien AA, Karamcheti V, Plevyak J (1993) The Concert System Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs. UIUC DCS Tech Rep R-93-1815
12. Grimshaw AS, Weissman JB, Strayer WT (1996) Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. ACM Trans Comput Syst (TOCS), Vol. 14, Issue 2, pp. 139170
13. Benson GD, Olsson RA (1997) A Portable Run-Time System for the SR Concurrent Programming Language. Workshop Run-Time Syst Parallel Program (RTSPP)
14. Victor Malyshkin and Vladislav Perepelkin. Optimization methods of parallel execution of numerical programs in the LuNA fragmented programming system // The Journal of Supercomputing, Springer, Volume 61, Number 1 (2012), pp. 235-248.
15. S.E.Kireev. A Parallel 3D Code for Simulation of Self-gravitating Gas-Dust Systems // PaCT-2009 proceedings, Springer, LNCS 5698 (2009), pp. 406413 (http://ssd.sscc.ru/en/dlb)
16. Maltsev A.I. Algorithms and recursive functions / Maltsev A.I.; Translated from the first Russian ed. by Leo F.Boron, with the collaboration of Luis E.Sanchis, John Stillwell and Kiyoshi Iseki. - Groningen: Wolters-Noordhoff Pub. Co. - [1970]. - 372 p.

17. H. Rogers, Jr., 1967. The Theory of Recursive Functions and Effective Computability, second edition 1987, MIT Press. ISBN 0-262-68052-1 (paperback), ISBN 0-07-053522-1
18. S.Kireev and V.Malyshkin Fragmentation of numerical algorithms for parallel subroutines library - The J. of Supercomputing, Springer, Springer, Volume 57, Number 2 / August 2011 pp. 161-171
19. Kraeva MA, Malyshkin VE (2001) Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. Int J Future Gener Comput Syst, Elsevier Science. Vol. 17, No. 6, pp. 755765
20. Maxim Gorodnichev, Sergey Kireev and Victor Malyshkin. Optimization of inter-cluster communications in the NumGRID. In the Proceedings of the second Russia-Taiwan symposium on Method and tools of Parallel Programming Multicomputers (MTPP). Springer, LNCS series, Vol. 6083, pp. 78-85, 2010