Optimization of Parallel Execution of Numerical Programs in LuNA Fragmented Programming System

Victor Malyshkin and Vladislav Perepelkin

Institute of Computational Mathematics and Mathematical Geophysics
Russian Academy of Sciences
Prospekt Akademika Lavrentjeva 6, Novosibirsk, Russia
{malysh, perepelkin}@ssd.sscc.ru

Abstract. Organization of high performance execution of fragmented programs met the problem of choice of acceptable way of their execution. The possibilities of execution optimization on the stages of fragmented program development, compilation and execution are considered. The methods and algorithms of optimizations are suggested to be included both in fragmented programming language and in run-time system.

Key words: parallel programming, fragmented programming, high performance computing, program execution optimization.

1 Introduction

The idea of data and algorithms fragmentation is exploited in programming at least from the early 1970th [1–8]. Generally, the model of a program in this approach looks as follows. The data are fragmented and values of the simple (atomic) variables can be the data aggregates (*data fragment* (DF)) that usually reflect the essence of an object domain. For example, a cell of a 3D-mesh in Particle-In-Cell method can be considered as atomic part of the description of the minimal part of a simulated phenomenon. The variables' description can reflect the structure of its value, i.e., a DF structure. In particular, in numerical algorithms a sub-matrix of a matrix can be defined as a DF and the whole matrix is represented as an array of its sub-matrices. An operation computes the values of output variables from the input variables. An operation plus input and output variables is called a *fragment of computation* (FC) (Fig. 1). DFs have unique names that predefine single assignment mode of programming. Therefore, if two or more FCs compute the value of a certain variable, then the same DF is yielded by each FC as the value of this variable (fig. 1.b and 1.c). This is the restriction on the set of permitted interpretations. Any FC has also unique name. A *fragmented program* (FP) is represented as a computable set of FC. A FC can be executed once if certain values are assigned to all of its input variables. Formal definitions and more details can be found in [3].

Different modifications of this general model were embodied in programming systems as commercial [2, 4] or academic [5] product. In [2] instead of usually used

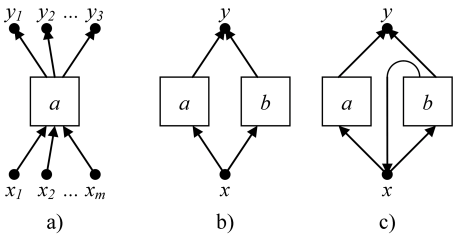

Fig. 1. Fragments of computation and data fragments.

run-time system for FCs execution a special operating system was developed. Many programming systems use run-time systems for computation organization [6–12]. Problems of the set of the FCs execution are well known and shortly can be formulated as:

a. dynamic resources allocation,
b. dynamic DFs and FCs distribution and their migration among the processor elements (PE) of a multicomputer,
c. dynamic choice of a certain FC for execution.

Obviously, the algorithms, used to overcome the problems above, cardinally influence on providing such important properties of a program as dynamic tunability to all the available resources, dynamic load balancing, data transfer in parallel with computations and so on.

Taking into account the necessity to solve the above listed problems, basing on the experience of the other related developments, we started the development of our fragmented programming system LuNA, based on the theory of parallel program synthesis [3]. The system is oriented to the development of FPs, implementing large scale numerical models. First planned application of LuNA is the creation of parallel numerical subroutine library. Every subroutine should be automatically provided with all the necessary dynamic properties. Aside from the different problems of the LuNA creation we are concentrated here on the problem of high performance execution of a FP.

2 LuNA model of a program.

The general model is modified in order to meet LuNA needs. LuNA is the programming system, not the system of program synthesis. Therefore, there is no necessity to include in the LuNA model the single assignment. In order to facilitate the resources allocation and the data distribution the variables are defined similar to variables definition in programming languages, i.e., the name of a variable denotes the memory extent where different values are kept at different moments.

Contrary to variable's names an operation name denotes a certain execution of an operation. In particular, if an operation (procedure) b should be applied to

every entry of the array x (fig. 2.a), then i-th execution of b is denoted as b^i . Two sets of the FCs are defined in fig. 2.a: $\{a^i, i = 1, 2, 3, \ldots\} = \{i = 1, 2, 3, \ldots | x[i] \rightarrow a^i \rightarrow x[i+1]\}$, $\{b^i, i = 1, 2, 3, \ldots\} = \{i = 1, 2, 3, \ldots | x[i] \rightarrow b^i \rightarrow y[i]\}$. Each FC is named by the indexed name of its operation.

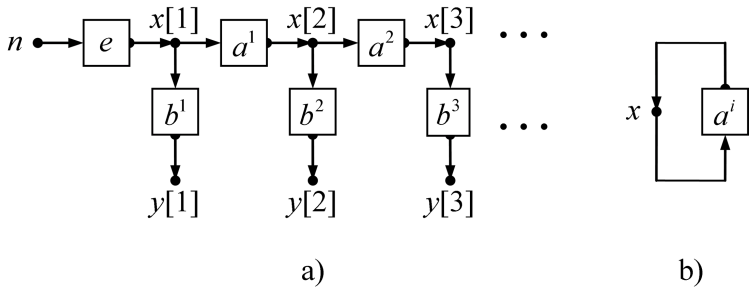

Fig. 2. Sample FP (a) and loop computations (b).

Loopwise computations are shown in the fig. 2.b. The set of the FCs $\{i = 1, 2, 3, \ldots | x \rightarrow a^i \rightarrow x\}$ is defined. The order of the FCs execution is defined by the binary partial order relation $\rho = \{i = 1, 2, 3, \ldots | < a^i, a^{i+1} >\}$ (*relation* ρ). This means, that FC a^i should be executed before a^{i+1}. For computation in fig. 2.a the order of the FCs execution is defined by the information dependencies between operations whereas for computations in fig. 2.b the order should be defined explicitly. The resources allocation for the array x implementation is done in fig. 2.b by a user.

A certain FC can be chosen for execution, if its execution does not contradict to the relation ρ. The relation ρ should guarantee the correct execution of the FP.

As result, LuNA language contains at least the facilities for definition of the DFs, the FCs and the relation ρ on the set of FCs. Therefore, the LuNA user has the possibility to define at least partially the resources allocation (see fig. 2.b). This might substantially improve the results of automatic resources allocation and the FCs distribution done by LuNA compiler and run-time system.

3 Fragmented algorithms and their execution.

3.1 Matrices multiplication.

Fragmented version of the algorithm of two square matrices A and B multiplication, $C = A \times B$, is considered. Matrices are represented by the square $K \times K$ matrices of square sub-matrices $A_{i,k}$, $B_{k,j}$, $C_{i,j}$ (see fig. 3). Sub-matrices $A_{i,k}$, $B_{k,j}$, $C_{i,j}$ are the DFs here.

Intermediate values are kept in DFs $C_{i,j,k}$. The FCs $\mathcal{F}_{i,j,k}$ and $\mathcal{S}_{i,j}$ perform matrices multiplication $A_{i,k} \times B_{k,j} = C_{i,j,k}$ and summation $C_{i,j} = \sum_{k=1}^{K} C_{i,j,k}$

respectively. Necessary order to compute the product C correctly is $\mathcal{F}_{i,j,k} < \mathcal{S}_{i,j} \forall i,j$.

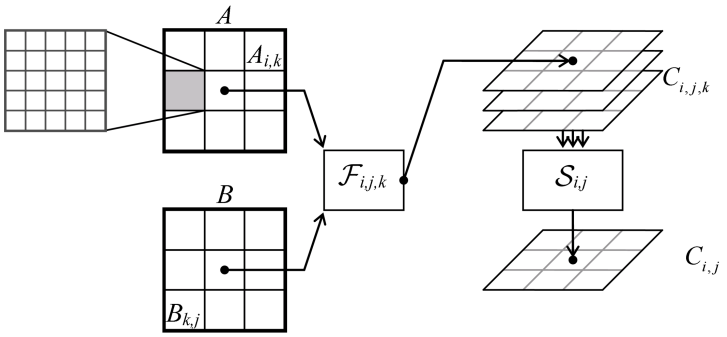

Fig. 3. Schema of fragmented algorithm of matrices multiplication

Run-time system chooses a certain FC for execution in any order, which does not contradict to the relation ρ . In this case the correct result of the FP execution will be produced, but the FP execution performance might be poor. For example, execution of any FC $\mathcal{F}_{i,j,k}$ produces a DF $C_{i,j,k}$, therefore some memory extent should be allocated to keep its value. On the other hand, after $\mathcal{S}_{i,j}$ execution the memory, allocated for the DFs $C_{i,j,k}$, is released. Run-time system should take this into account when FC is chosen for execution, otherwise the computer memory might be exhausted not productively. Good (recommended) order would be the one with FCs $\mathcal{S}_{i,j}$ executed as soon as possible (but only after all the $\mathcal{F}_{i,j,k}$ with the same i and j are finished).

Another problem here is data distribution. To what PE a certain DF should be allocated? Random distribution results in huge communications and some PEs can be idle due to load imbalance. In LuNA there are opportunities, which are considered in section 4, to control the DFs distribution and migration to provide good performance.

3.2 LU-factorization.

Another example is the fragmentation of LU-factorization algorithm. Square $n \times n$ matrix A is factorized into lower triangular matrix L and upper triangular matrix U, $A = L \times U$.

$$\begin{array}{l} u_{1,j} = a_{1,j}, j = 1, \ldots, n \\ l_{j,1} = \frac{a_{j,1}}{u_{1,1}}, j = 2, \ldots, n \\ u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} u_{k,j}, i = 2, \ldots, n; j = i, \ldots, n \\ l_{j,i} = \frac{1}{u_{i,j}}(a_{j,i} - \sum_{k=1}^{i-1} l_{j,k} u_{k,j}) i = 2, \ldots, n; j = i+1, \ldots, n \end{array} \quad (1)$$

Matrix A is represented by the $K \times K$ matrix of $A_{i,j}$ sub-matrices. Matrices L and U are both represented by the same $K \times K$ matrix of $L_{i,j}$ and $U_{i,j}$ sub-matrices (fig. 4.a). They are the DFs. Each DF $A_{i,j}$ is processed in accordance to formulas (1) by the FCs $\mathcal{L}_{i,j}$, $\mathcal{D}_i$ or $\mathcal{U}_{i,j}$, for lower diagonal, diagonal and upper diagonal DFs respectively, as shown in fig. 4.a. The relation ρ is defined by the information dependencies in (1) as follows: The FCs $\mathcal{D}_i$ should be executed after $\mathcal{L}_{i,j-1}$ and $\mathcal{U}_{i-1,j}$, the FCs $\mathcal{U}_{i,j}$ should be executed after $\mathcal{U}_{i-1,j}$ and $\mathcal{D}_i$, and the FCs $\mathcal{L}_{i,j}$ should be executed after $\mathcal{L}_{i,j-1}$ and $\mathcal{D}_j$ (fig 4.b).

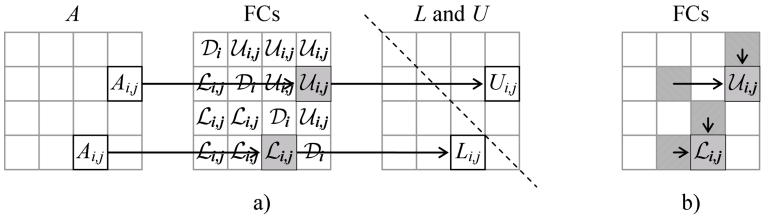

Fig. 4. LU factorization. General schema (a) and order schema (b).

It is clear from 4.b, that execution of the FC $\mathcal{D}_i$ increases the number of the FCs ready to be executed (*ready FC*), whereas the execution of the other FCs reduces it. To exploit the algorithm's parallelism the set of the ready FCs must be large enough to load all the PEs over time. Therefore, run-time system should provide the execution of the FCs $\mathcal{D}_i$ before the other FCs' execution. The fig. 5 illustrates two FC choice algorithms. The first is to choose the $\mathcal{D}_i$ FCs last. It leads to situations, when only one FC is ready (fig. 5.a), and all the PEs are idle waiting for only one FC to be completed. The second algorithm chooses the FCs according to the diagonal front, as shown in fig. 5.b. It provides more parallelism and enables more effective program execution. The opportunities to control the FC choice are considered in section 4.

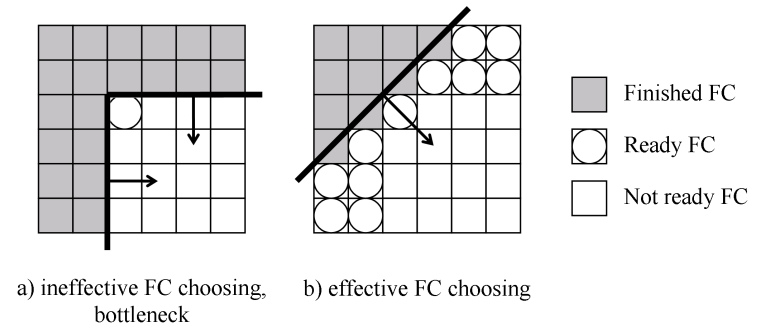

Fig. 5. LU factorization, FC choosing.

4 The other opportunity to optimize the execution of a set of FC.

There are several technological opportunities, used in LuNA system, to improve a set of FC execution. Those are the means to express supplementary information about a FP and recommended ways of its execution. Note, that different hardware requires different recommendations; therefore both general and hardware-dependent recommendations can be provided by user. Run-time system selects the most suitable recommendation set.

4.1 Priority.

A real number called *priority* is assigned to each FC. At any moment, run-time system tries to choose for execution a fragment with the highest priority available. This allows controlling the FC execution flow to reach better resources usage. The LuNA run time system not only chooses the highest priority FC for execution, but also schedules FCs execution in such a way, that high priority FCs become ready sooner. In the LU-factorization FP (example in section 3.2 above) the DFs $\mathcal{D}_i$ should have higher priorities, than the rest of the FCs.

4.2 Groups and derivation algorithm use.

LuNA language has the facilities in order a user could describe a *group* of FCs. Usually, the information depended FCs are included into the group. The FCs, belonging to such a group, are executed in accordance with the MGF strategy (Member of Group First). With the MGF strategy if a certain FC, included into a group, was chosen for execution, then the higher priority is assigned to all the other FCs, belonging to the same group. This strategy leads to consumption of the intermediate DFs soon after they were computed.

The groups can be formed by LuNA compiler using the derivation algorithm [3]. This algorithm processes the countable set of the FCs, that constitute the FP, re-constructs the set of functional terms, implemented by the FP, and then folds them into the finite sets of indexed functional terms (see fig. 2.b). A range of different optimizing transformations of the sets of indexed functional terms is also provided. Any FCs, included into a certain indexed functional term, is also included into a group. Construction of these sets of indexed functional terms permits the use automatically the MGF strategy in run-time system.

In the matrices multiplication algorithm above (section 3.1) in order to optimize the resources usage the groups' definition can be exploited. All the FCs $\mathcal{F}_{i,j,k}$ with the same values of their indices i and j are included into the same group. If a certain FC is chosen for execution, the priorities of all the other FCs from its group are increased. Thus, mostly this group's FCs will execute. As result, all the intermediate resources keeping the DFs $D_{i,j,k}$ will be released.

4.3 Weight of FC.

Weight of FC is a real-valued function, defined on the set of FCs. It represents an estimation of the FC's execution time. In the LU-factorization (section 3.2) the time of computation of the FCs increases in the direction to the right bottom corner of the matrix. The value of Weight gives the run-time system this info.

4.4 Neighborhood relation.

A binary neighborhood relation ν is defined on the set of the 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 the DFs, which are the input values of a certain FC, and location of them in the same PE leads to reduction of the total communication overhead. This relation can be constructed automatically, basing on the structure of information dependences of the FP, but in general case, neighborhood relation ν is better defined by the user.

Neighborhood relation ν is considered when performing initial data distribution with low communication overhead, or dynamic load balancing, that also should keep the neighborhood relation ν .

The numerical algorithms employ limited number of spatial data structures, like vectors, matrices, arrays, 3D meshes. LuNA supports explicit declaration of such data structures and implements a number of algorithms to perform initial distribution and structure-keeping dynamic load balancing on commonly used hardware network topologies, like 3D torus, cluster or complete graph.

5 Performance tests.

The ideas presented were implemented in experimental LuNA functional programming system. It comprises the language of FPs description, the translator to an executable representation and the run-time system. A number of tests was performed. Priority and group testings were performed on a 8-core SMP multiprocessor. Weights and neighborhood relation testings were performed on a cluster.

5.1 Priority testing.

The test should demonstrate the advantages of the priority use. Three tests were accomplished for LU-factorization (example in section 3.2):

1. *Ineffective.* The relation ρ is defined in such a way, that the ineffective order of the FCs execution would be implemented (fig. 5.a.).
2. *Priority-based.* The relation ρ reflects only information dependences between the FCs. Two different priorities were assigned to the FCs. The higher priority was assigned to $\mathcal{D}_i$ FCs and the lower for the rest of the FCs. Certain order was chosen dynamically by the run-time system.

3. *Effective.* The relation ρ is defined in such a way, that the effective order of the FCs execution would be implemented (fig. 5.b)

The results of testing are shown in fig. 6.a.

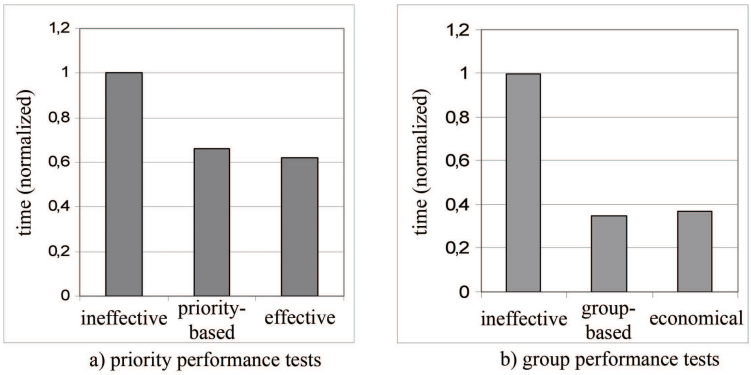

Fig. 6. Priority and group performance tests.

5.2 Group testing.

Groups' use influences to the execution time of the matrices multiplication program (example in section 3.1) was tested in 3 tests:

1. *Ineffective.* The relation ρ is such defined that all the FCs $\mathcal{S}_{i,j}$ are executed the last. As a result all the DFs $C_{i,j,k}$ are kept in the memory long time. This is the most time- and memory-consuming FP execution.
2. *Group-based.* The priorities of the FCs of the same group are dynamically increased.
3. *Economical.* The FCs execution from another group is never started before the execution of all the FCs from a currently executed group are completed.

The results are shown in fig. 6.b.

5.3 Neighborhood relation and FC weight testing.

The model of fragmented algorithm of Particle-In-Cell method (PIC) [15] implementation was used for testing. This is explicit finite differences 3D scheme. A 3D mesh is represented by 3D grid of DFs. Processing of each DF requires the values from its 26 neighbors. The 3D grid of the DFs is processed iteratively by the FC $\mathcal{F}^t_{i,j,k}$, where i, j and k are the indices of the FCs name, and t is the iteration number. The execution time of the FC $\mathcal{F}^t_{i,j,k}$ is defined by the function $f_{i,j,k}(t)$. Different definitions of $f_{i,j,k}(t)$ lead to different model behaviors. The

$f_{i,j,k}(t)$ was chosen in such a way that PIC model imitated the soliton orbiting a massive center. Correspondingly the time of the FCs $\mathcal{F}^t_{i,j,k}$ execution was changing.

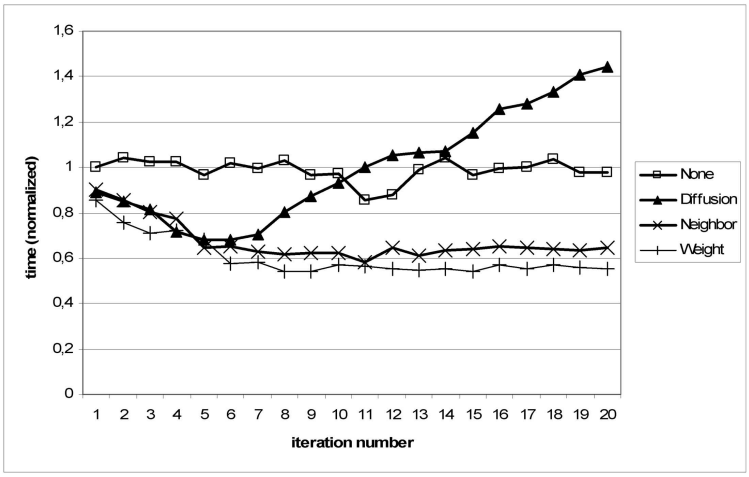

Fig. 7. Neighborhood relation influence on dynamic load balancing.

On the graphics four balancing versions are shown. The abscissa axis is the iteration number, the ordinate axis is the iteration execution time.

The *None* version has no dynamic load balancing, the DFs don't migrate. The execution time remains about the same, but a lot of PEs' time is wasted, since the workload is not uniformly distributed.

The *Diffusion* version is a certain diffusion dynamic load balancing algorithm. The load is being balanced, but the communication overhead grows, since the DFs are mixing up, and the total execution time even exceeds the unbalanced version.

The *Neighbor* variant is the diffusion load balancing with neighborhood relation taken into account. Those DFs migrate, which have more neighbors in the target PE. Such load balancing keeps communication overhead at certain level, and it doesn't grow with the lapse of time.

The *Weight* variant is the same as the *Neighbor*, but the FCs' weights are taken into account by run-time system. The function $f_{i,j,k}(t)$ is used as the FCs' weight. The load balancing algorithm works more accurate, compared to Neighbor variant, and the execution time is a bit less.

6 Conclusion.

The LuNA system of fragmented programming is yet under development and improvement. Our next step is the development of the parallel numerical subroutine library on the basis of LuNA system.

References

1. Glushkov, V.M., Ignatiev, M.V., Myasnikov V.A., Torgashev V.A.: Recursive machines and computing technologies. In: IFIP Congress, Vol.1., pp. 65–70. North-Holland Publish. Co (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.A., Malyshkin V.E.: 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, Vol. 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: 9th International conference on Parallel Computing Technologies (PaCT-2007), Springer Verlag, LNCS, Vol. 4671, 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, Vol. 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), Vol. 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: Workshop on Run-Time Systems for Parallel Programming (RTSPP) (1997)
13. Malyshkin, V.E., Sorokin, S.B, Chauk, K.G.: Fragmentation of numerical algorithms for the Parallel Subroutine Library. Springer Verlag, LNCS, Vol. 5698, pp. 331–343 (2009)
14. Kraeva, M.A., Malyshkin, V.E.: Implementation of PIC Method on MIMD Multicomputers with Assembly Technology. In: HPCN Europe 1997, Springer Verlag, LNCS, Vol. 1277, 1997. pp. 541–549 (1997)
15. Kraeva, M.A., Malyshkin, V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. In: Int. Journal on Future Generation Computer Systems, Elsevier Science. Vol. 17, No. 6, pp. 755–765 (2001)