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

Victor Malyshkin and Vladislav Perepelkin

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

{malysh, perepelkin}@ssd.sscc.ru

http://ssd.sscc.ru

Abstract. The organization of high performance execution of a fragmented program has encountered with the problem of choosing of an acceptable way of its execution. The potentialities of optimizing the execution at the stages of fragmented program development, compilation and execution are considered. The methods and algorithms of such an optimization are proposed to be included into the LuNA fragmented programming language, compiler, generator and run-time system.

Keywords: parallel programming, fragmented programming, high performance computing, program execution optimization

1 Introductory definitions and relative works

The idea of data and algorithms fragmentation has been exploited in programming, at least, since the early 1970-s [1–8]. Different modifications of this approach were embodied in programming systems [3–5]. Many programming systems use the run-time systems for the organization of computation [6–12]. In [3], instead of a commonly used run-time system for organization of the program execution, a special hardware and operating system were developed. Our LuNA fragmented programming system project is oriented to the creation of a parallel numerical subroutine library.

A general model of a program in the above-mentioned systems can be described as computational model [1].

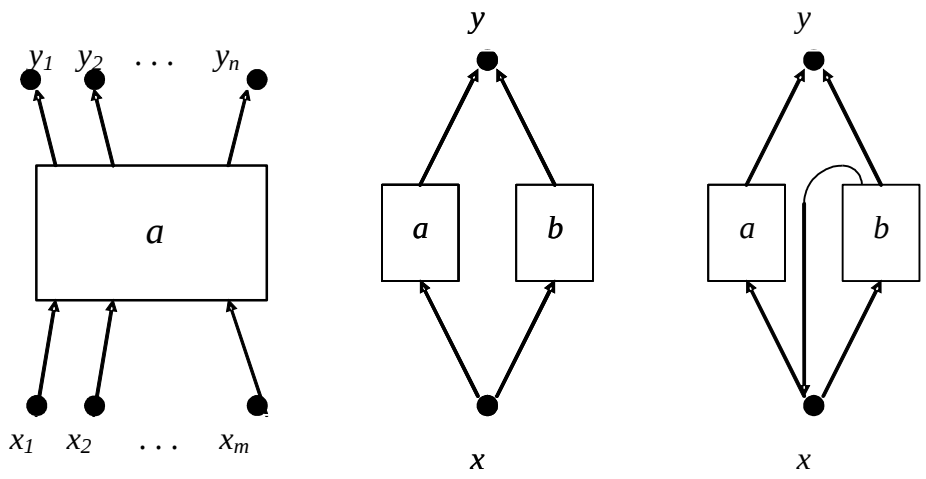
General model definition

Given:

- The finite set $\mathbf{X}=\{x, y, ..., z\}$ 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 $a \in \mathbf{F}$ describes the possibility to compute the variables out(*a*) from the variables in(*a*), for example, with the use of a certain procedure. The model can be graphically depicted (Fig. 1)



a) b) c)

Fig. 1 Examples of operations, variables and model

Let $V \subseteq \mathbf{X}$, $F \subseteq \mathbf{F}$ 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 $\forall i(x_i \in \text{out}(t^i))$, $\text{in}(t) \bigcup_{i=1}^{s} \text{in}(t^i)$, 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.

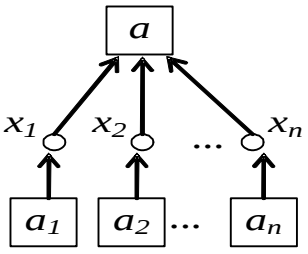


Fig. 2 A depicted term

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

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 functional terms is considered as a representation of an algorithm.

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

- 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 $a \in \mathbf{F}$, $\text{in}(a)=\{x_1, x_2, ..., x_m\}$, $\text{out}(a)=\{y_1, y_2, ..., y_n\}$, 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, $\text{in}(a)=\{x_1, x_2, ..., x_m\}$, $\text{out}(a)=\{y_1, y_2, ..., y_n\}$, then $\boldsymbol{I}(\text{out}(a))=val(t)=(d_1,d_2,...,d_n)=f_a(val_{x1}(t_1),val_{x2}(t_2),...,val_{xn}(t_n))$.

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.

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

For definition of mass computations this model should be extended by inclusion of indexed operations and indexed variables (arrays). This technical work can be easily done. Obviously, in this extended model, a mass algorithm is represented by an infinite recursively countable set of functional terms.

A program that implements an algorithm, represented by a set of functional terms, can be constructed with the procedure calls to mod_a in the order not

contradicting to the information dependences between the operations imposed by the terms structure. Usually, a run-time system is used to implement all the calls in a proper order.

2. The Potentialities of improving the algorithm execution performance

The algorithm representation as a set of functional terms does not automatically provide the algorithm execution with a good performance. Therefore in the LuNA model of a program some reductions of the general model should be done.

Problems of the efficient algorithms execution are well known and in brief can be formulated as:

a. Folding of a countable set of functional terms, representing an algorithm, into a finite representation.
b. Dynamic allocation of a multicomputer resources.
c. Dynamic data and operations distribution and their migration among the processor elements (PE) of a multicomputer.
d. Dynamic choice of a certain operation for execution.

The two main reductions are made for the LuNA model of a program. These are data and operations aggregation/fragmentation and multiple assignments.

Taking into account the need for solving the above-listed problems, based on the experience of the other related developments, including our experience gained in the large scale numerical models development [15], we started the development of our fragmented programming system LuNA, based on the general model. The system is oriented to the development of fragmented programs, implementing numerical models.

The first proposed LuNA application is the creation of a parallel numerical subroutine library. Every subroutine should be automatically provided with all the necessary dynamic properties. Aside from different problems of the LuNA creation, we concentrate here on the problem of the fragmented programs high performance execution. The proposed LuNA programming system is oriented to the distributed memory supercomputers, where computation nodes can be the shared memory multiprocessors. The network topology between nodes may be a tree, a mesh, a torus, etc.

In comparison with other libraries of parallel numerical subroutines, the LuNA library has a number of advantages. First, the LuNA library is highly portable. Subroutines are automatically tuned to available hardware resources by the LuNA compiler. Their dynamic properties, like dynamic load balancing, are automatically provided by the run-time system. Porting the LuNA library to another platform requires no changes of the library subroutine texts, though it requires the development of a new run-time system for a new platform. The LuNA run-time system is oriented to execution of numerical subroutines. This specialization is reflected in the algorithms designed for implementation of the LuNA run-time system, which are mostly oriented to minimization of a subroutine execution time.

The second advantage is the ability of global optimization of multiple subroutines executed in parallel. In this case, the LuNA run-time system considers all the subroutines inputs, outputs and intervenient data as common memory. This allows avoiding unnecessary synchronizations between calls of subroutines and improving the resources distribution between subroutines. Thirdly, the programmer does not has to program communications, synchronizations and resources management. Instead, he/she only has to specify the program behavior.

A class of problems, that can be well programmed in the LuNA depends on the algorithms implemented in it. Our current implementation is oriented to problems with massive parallelism, regular data and computations structure, such as iteration processes on regular meshes, matrix and vector operations, etc. Other classes of problems can also be programmed in the LuNA, but the performance of their execution may not be satisfactory.

2.1.Variables, operations and data fragmentation

Variables and operations of the general model can be aggregated. Therefore, the values of simple (atomic) variables can be the data aggregates. The aggregates of variables and data both are denoted as *data fragments* (DF) that usually reflect the essence of an object domain. For example, a cell of a 3D-mesh in the Particle-In-Cell method can be considered as atomic part (DF) of the description of a minimal semantic part of a simulated phenomenon. In numerical algorithms a sub-matrix of a matrix can be defined as a DF, and the whole matrix is represented as a 2D array of its sub-matrices. An aggregated operation plus its

input and output DFs is called a *computation fragment* (CF). The DFs and CFs have unique names that predefine a single assignment mode of programming. A *fragmented program* (FP) is represented as a computable set of CFs. A CF can be once executed if certain values are assigned to all of its input DFs.

Consider one simple example of the algorithm of summation of the two vectors $z=x+y$ fragmentation (Fig. 3.a). Let $n=M\times N$, n be the number of entries of the vectors x, y and z; M is the number of aggregated entries (the number of entries in a fragments), N is the number of fragments. As is shown in Fig. 3.b, the variables x_i, y_i and z_i are aggregated into DFs X_I, Y_I and Z_I, respectively, where $X_I=\{x_i|i=I\times M,\ldots,(I+1)\times M-1\}$. Operations f_i are similarly aggregated: CF $F_I=\{f_i|i=I\times M,\ldots,(I+1)\times M-1\}$. Such an algorithm is called *fragmented algorithm* (FA). The algorithm fragmentation is studied in a number of projects, in particular [13, 14].

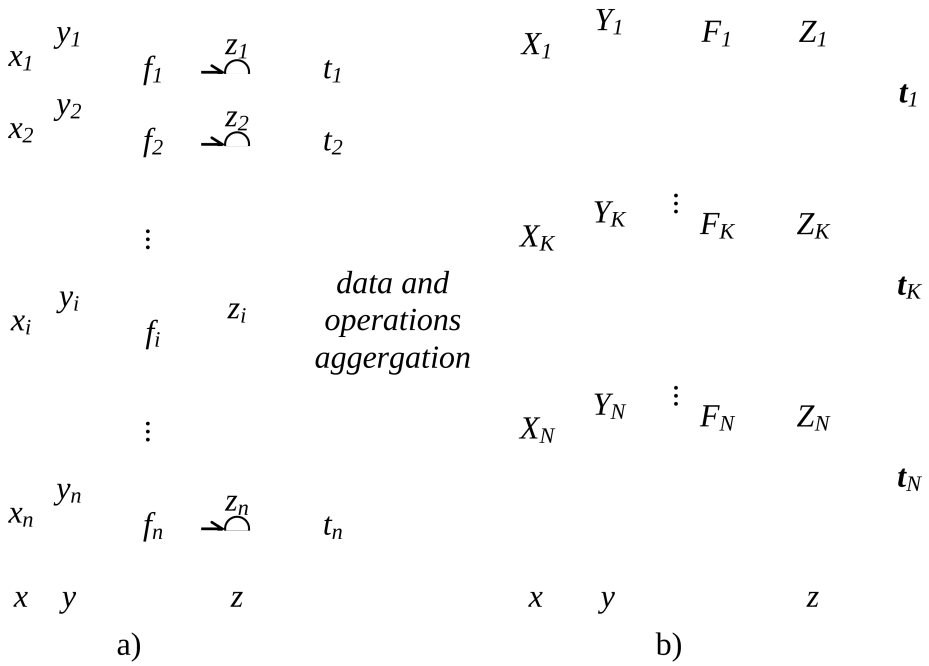


Fig. 3 Fragmentation of the algorithm of summation of entries of two vectors

Variables, operations and data fragmentation is done in order to make a decision on resources allocation or the operation choice for execution of an aggregated object, but not of every entry separately. Such a fragmentation substantially reduces the overhead. As a rule, such a fragmentation should satisfy the rules of *time* and *spatial locality*.

DF *spatial locality* means that the DF elements are jointly processed, by the same procedure.

DF *time locality* means that the DF processing is restricted to a limited time interval.

CF *spatial locality* means thet its operations process variables from a limited number of DFs.

CF *time locality* means that all *CF* operations are executed within a limited time interval.

2.2. Multiple assignment

Multiple assignments of DFs is permitted, so DFs are able to keep different values at different moments similar to variables of an imperative programming language. Strictly speaking, a multiple assignable DF is a union of single assignment DFs, which will be mapped onto the same slot of memory.

Multiple assignment is a facility to construct the resources allocation and to make (with the user's help) the folding of an infinite set of functional terms into the finite FP. Multiple assignment is used for the implementation of numerical models in imperative programming languages. For example, iterative processing of a finite differences scheme is implemented in the same memory extent.

Permission of multiple assignments requires an additional control in order to provide the correct use of different values. Consider an example (Fig. 4): DFs X_i are processed by CFs A_i. in(A_i) = {X_{i-1}}, out(A_i) = {X_i} (Fig. 4.a). All X_i are assigned for implementation into the same multiple assignments DF X (Fig. 4.b). If no additional control is defined, a value, produced by CF A_i may be consumed by any other A_j, while it has to be consumed by A_{i+1}. To solve this problem, the order of CFs execution is defined not by information dependences, but explicitly by the partial order relation ρ, defined on the set of CFs. In particular, ρ should contain the entries $\{\langle A_{i-1},A_i\rangle|\forall i=1,\ldots,N\}$.

a) X_0 $\boxed{A_1}$ X_1 A_2 ... A_N X_N

b) X A_i $\forall i\langle A_{i-1},A_i\rangle\in\rho$

Fig. 4 Definition of CF execution order: a) information dependences, b) partial order relation

Consider another example (Fig. 5). Two DFs: X_1 and X_2 are assigned for implementation into the same multiple assignments DF X. If no additional control

is defined, a value, produced by CF A_1 may be replaced by that, produced by CF A_2 before the first one is consumed by CF B_1. To solve this problem, the direct control should be defined to provide the correct use of DFs' values (in particular, a semaphore can be used).

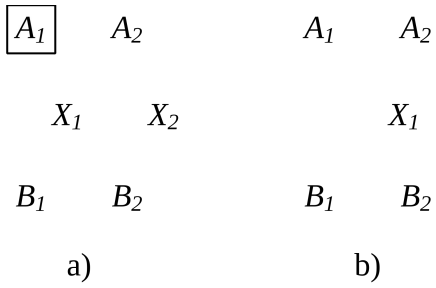


Fig. 5 Multiple assignment DFs.

In such a way, the *LuNA fragmented program* (LFP) is the five-tuple ⟨**DF**, **CF**, ρ, **DC**, *Code*⟩, where **DF** is a set of all the DFs, **CF** is a set of all the CFs, $\rho \subseteq$ **CF**×**CF** is the partial order relation, **DC** is a direct control. The *Code* function assigns the mod_a (*fragment of code*) for every *CF*, $\forall a \in$ **CF** $Code(a)=mod_a$.

A minimal partial order relation $\rho_{\min} \subseteq$ **CF** ×**CF** contains all the entries imposed by the information dependences between operations of the algorithm. Adding new entries into ρ, a set of possible ways of the LFP execution can be reduced, which is very important for the LFP execution optimization.

3.Examples of numerical algorithms fragmentation and the problems of their efficient execution

3.1. Matrices multiplication

A fragmented version of the algorithm of the two square $N \times N$ matrices A and B multiplication, $C=A \times B$, is considered. 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. 6). Here the sub-matrices $A_{i,k}$, $B_{k,j}$, $C_{i,j}$ are the DFs, $N=K \times M$.

The DFs $C_{i,j,k}$ are intermediate variables. The 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. Information dependences are described by the relation ρ that contains the pairs i,j,k ($F_{i,j,k}<S_{i,j}$).

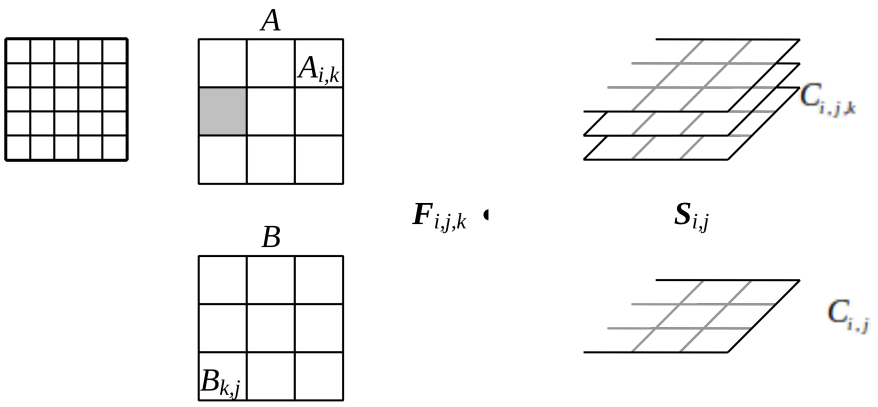


Fig. 6 Scheme of the fragmented algorithm of matrices multiplication

The run-time system chooses a certain CF for execution in any order, which does not contradict to the relation ρ. In this case, a correct result of the LFP execution will be produced, but the LFP execution performance might be low. For example, execution of any CF $F_{i,j,k}$ produces a DF $C_{i,j,k}$, therefore some memory extent should be allocated to keep its value. On the other hand, after $S_{i,j}$ execution, the memory, allocated for the DFs $C_{i,j,k}$, is released. The run-time system should take this into account when CF is chosen for execution, otherwise the computer memory might be exhausted unproductively. A good (recommended) order would be the one with CFs $S_{i,j}$ executed as soon as possible (but only after all $F_{i,j,k}$ with the same i and j are finished).

Another problem here is data distribution. To what PE a certain DF should be assigned for processing? A random distribution results in a huge communications overhead and load imbalance. In the LuNA, there are potentialities to control the DFs distribution and migration in order to provide a good performance of LFP execution.

3.2. LU-factorization

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

$$
\begin{aligned}
u_{1,j} &= a_{1,j},\ j = 1,\text{K}\ ,n \\
j_{j,1} &= \frac{a_{j,1}}{u_{1,1}},\ j = 2,\text{K}\ ,n \\
u_{i,j} &= a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} u_{k,j},\ i = 2,\text{K}\ ,n;\ j = i,\text{K}\ ,n \\
j_{j,i} &= \frac{1}{u_{i,j}}(a_{j,i} - \sum_{k=1}^{i-1} l_{j,k} u_{k,j}),\ i = 2,\text{K}\ ,n;\ j = i+1,\text{K}\ ,n
\end{aligned}
\qquad (1)
$$

Matrix A is fragmented and represented as $K \times K$ matrix of $A_{i,j}$ sub-matrices. The matrices L and U are both represented as $K \times K$ matrix of $L_{i,j}$ and $U_{i,j}$ sub-matrices (Fig. 7.a). They are the DFs. Each DF $A_{i,j}$ is processed according to formulas (1) by the CFs $L_{i,j}$, D_i or $U_{i,j}$, for the lower triangular, diagonal and upper triangular DFs, respectively, as shown in Fig. 7.a. The relation ρ is defined by the information dependences in (1) as follows: CFs D_i should be executed after $L_{i,j-1}$ and $U_{i-1,j}$, CFs $U_{i,j}$ should be executed after $U_{i-1,j}$ and D_i, and CFs Li,j should be executed after $L_{i,j-1}$ and D_j (Fig 7.b).

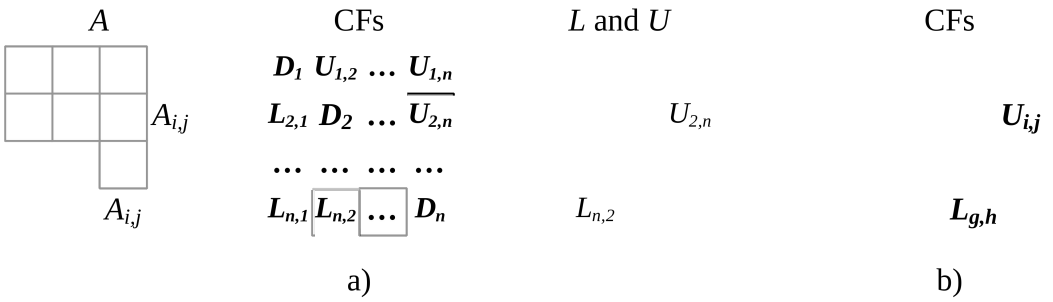


Fig. 7. LU factorization. General scheme (a) and order scheme (b)

It is clear from Fig. 7.b that execution of CF D_i increases the number of CFs ready for execution (*ready CFs*), whereas execution of other CFs reduces it. A set of the ready CFs should be wide enough to permanently load all the PEs. Therefore, the run-time system should provide execution of CFs D_i before the other CFs' execution. Fig. 8 illustrates two variants of CF choosing algorithms. The first thing is to permanently choose the D_i CFs in the last turn. This leads to the situations, when only one CF is ready (Fig. 8.a), and all the PEs are idly waiting for execution of one CF to be completed. The second algorithm chooses the CFs according to the diagonal front, as shown in Fig. 8.b. It provides a higher parallelism and a more efficient program execution.

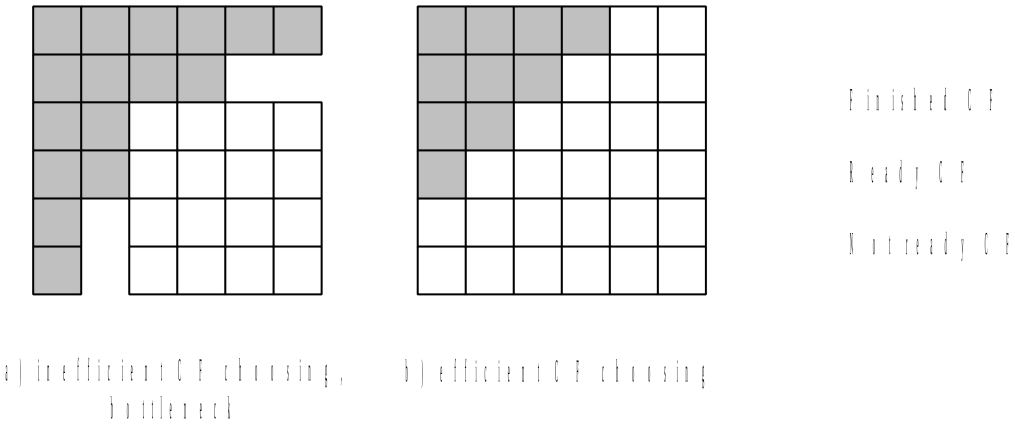


Fig. 8 LU factorization, CF choosing

4. The LuNA programming system

In the LuNA programming system, an algorithm passes through a number of stages of transformation in order to become an executable program. Initially, an algorithm is a set of functional terms and can be executed ‘as is’, like it is done in functional languages. In this case, the run-time system has to assign a resource to every object (CF and DF) and to organize execution according to the relation ρ. This is a complex procedure, which results in a low performance.

In the LuNA programming system (Fig. 9), the FA execution is divided into three main steps. 1) FA is statically analyzed and the basic LFP execution schedule is constructed by a compiler. 2) The generator includes into LFP the info on hardware properties. 3) The resources allocation and the order of CFs execution selection are dynamically performed by the run-time system, which performs the LFP execution according to the statically constructed schedule. It’s necessary to use the run-time system, because a part of decisions on the LFP execution can be taken dynamically only, for example, the resources allocation.

Fig. 9 Structure of LuNA programming system

The FA description consists of the two parts. The first one contains the description of the FA in the LuNA programming language, which is close to a mathematical description of the FA. The second part contains a source file in a conventional programming language (i.e. C++), in which the procedure definitions, implementing the FA’s code fragments, are included. Then the

programmer compiles the two parts into the C++ source file, which can be then compiled by a conventional compiler into an executable file.

5. The other potentialities of optimizing the LFP execution

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

5.1 Priority

A real number called *priority* can be assigned to each CF. In the course of LFP execution, the run-time system tries to choose for execution a fragment with the highest value of priority. This allows controlling the CF execution flow to attain a better resources usage. The LuNA run-time system not only chooses the highest priority CF for execution but also schedules the CFs execution in such a way, that high priority CFs become ready sooner. In the LU-factorization, LFP (see the above example) the DFs D_i have higher priorities than the rest of the CFs.

5.2 The use of groups and derivation algorithm.

The LuNA language has facilities for the user to be able to describe a *group* of CFs. Usually, the information-dependent CFs are included into a group. The CFs, belonging to such a group, are executed according to the MGF strategy (Member of Group First). With the MGF strategy if a certain CF, included into a group, was chosen for execution, a higher priority is assigned to all the other CFs, belonging to the same group. This strategy brings about the consumption of intermediate DFs' values soon after they were yielded.

Groups can be formed by the LuNA compiler using the derivation algorithm [3]. This derivation algorithm processes a countable set of the CFs defined by an algorithm implemented by the LFP, re-constructs these functional

terms, and then folds them into the finite sets of indexed functional terms (see Fig. 4.b). A range of different optimizing transformations of the sets of indexed functional terms is also provided. Any CF, included into a certain indexed functional term, is also included into a group. The construction of these sets of indexed functional terms permits one to automatically use the MGF strategy in the run-time system.

In the above matrices multiplication algorithm, in order to optimize the resources use, the explicit groups definition can be exploited. All the CFs $F_{i,j,k}$ with the same values of their indices i and j are included into the same group. If a certain CF is chosen for execution, the priorities of all the other CFs from its group are increased. Thus, all the CFs from this group will be executed. As a result, all the intermediate resources keeping the DFs $C_{i,j,k}$ will be soon released.

5.3 The CF weight

Weight of a CF is a real value. It represents an estimation of the CF's execution time. In the LU-factorization, the time of execution of the CFs increases towards the right bottom corner of the matrix. The value of *Weight* is calculated by the run-time system to optimize the next LFP execution.

5.4 The neighborhood relation

A binary neighborhood relation ν is defined on a 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 variables of a certain CF, and their location in the same PE leads to reduction of the total communication overhead. The relation ν can be automatically constructed, based on the structure of information dependences of the LFP, but in the general case, the neighborhood relation ν is better defined by the user.

The neighborhood relation ν is taken into account when the initial data distribution with a low communication overhead, or a dynamic load balancing are constructed keeping the neighborhood relation ν.

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

5.5. The LFP profiling

The LFP profiling, i.e. gathering information, while LFP is executed, is a valuable source of optimization information that can be used by a compiler, a generator and a run-time system. The profile comprises information on CFs execution times, PEs load while LFP is executed, DFs distribution information, effective network bandwidth, etc. This information is taken into account in scheduling of the next LFP execution. Profiling leads to reduction of the execution time for each next LFP launch (of course, up to some limit). The way of LFP execution may be different for different input data, therefore the profile in not always helpful.

6. Performance tests

The concepts presented were implemented in the experimental LuNA fragmented programming system. It comprises the language of the LFP description, the compiler to an executable representation, the generator and the run-time system. A number of tests were performed. Priority and group testing were executed on an 8-core SMP multiprocessor. The weights and the neighborhood relation tests were executed on a cluster.

6.1 Priority testing

This test should demonstrate advantages of the priority use. Three tests were accomplished for the LU-factorization:

1. *Inefficient*. The relation ρ is defined in such a way that the inefficient order of the CFs execution would be implemented (Fig. 8.a.).
2. *Priority-based*. The relation ρ reflects only the information dependences between the CFs. A higher priority was assigned to D_i CFs and the lower – for the rest of the CFs. A certain order was dynamically chosen by the run-time system.

3. *Efficient*. The relation ρ is defined in such a way that the efficient order of the CFs execution would be implemented (Fig. 8.b)

The results of testing are shown in Fig. 10.a. The time is normalized on 0 to 1 scale, where 1 corresponds to *inefficient* execution time).

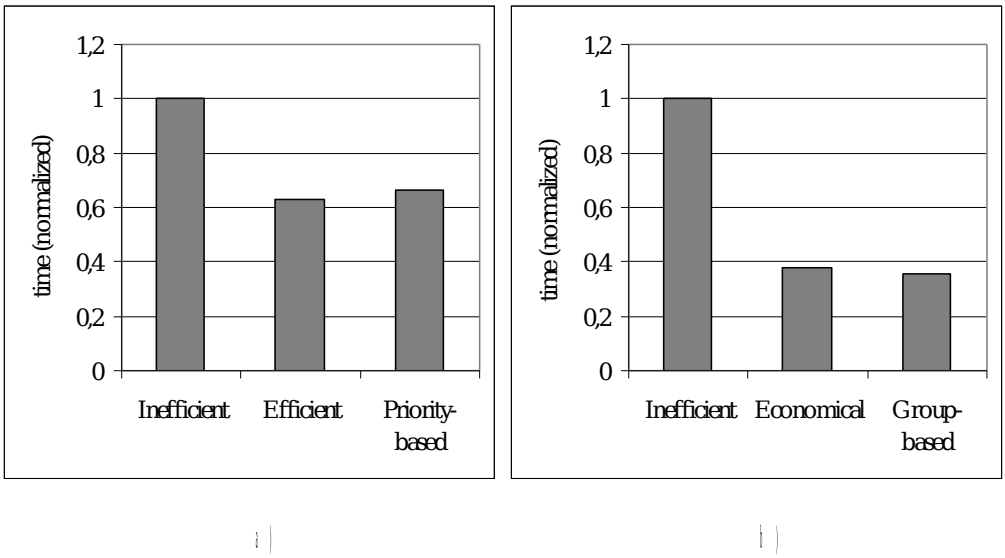


Fig. 10 Priority and group performance tests

6.2 Group testing.

The manner of how groups affect the execution time of the matrices multiplication program was tested in 3 tests:

1. *Inefficient*. The relation ρ is defined so that all CFs $S_{i,j}$ are the last to be executed. As a result, all DFs $C_{i,j,k}$ are kept in the memory long time. This is the most time- and memory-consuming LFP execution.

2. *Group-based*. The priorities of the CFs of the same group are dynamically increased.

3. *Economical*. The CF execution from other groups has never started before the execution of all the CFs from a currently executed group are completed. The test demonstrates the most memory-saving way of the LFP execution among the 1–3 tests.

The results are shown in Fig. 10.b. The time is normalized on a 0 to 1 scale, where 1 corresponds to *inefficient* execution time).

6.3 The neighborhood relation and CF weight testing

The model of a fragmented algorithm, implementing Particle-In-Cell method (PIC) application to the solution of an astrophysical problem[15] was chosen for testing. This is an explicit finite difference 3D scheme. A 3D mesh is represented by a 3D grid of DFs. Processing of each DF requires values from its 26 neighbors. A 3D grid of the DFs is processed iteratively by CF $F_{i,j,k}^{t}$, where i, j and k are indices of the CFs name, and t is the iteration number. The execution time of the CF $F_{i,j,k}^{t}$ 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 the PIC model imitated a soliton orbiting a massive center. Respectively, the time of the CFs $F_{i,j,k}^{t}$ execution was changing.

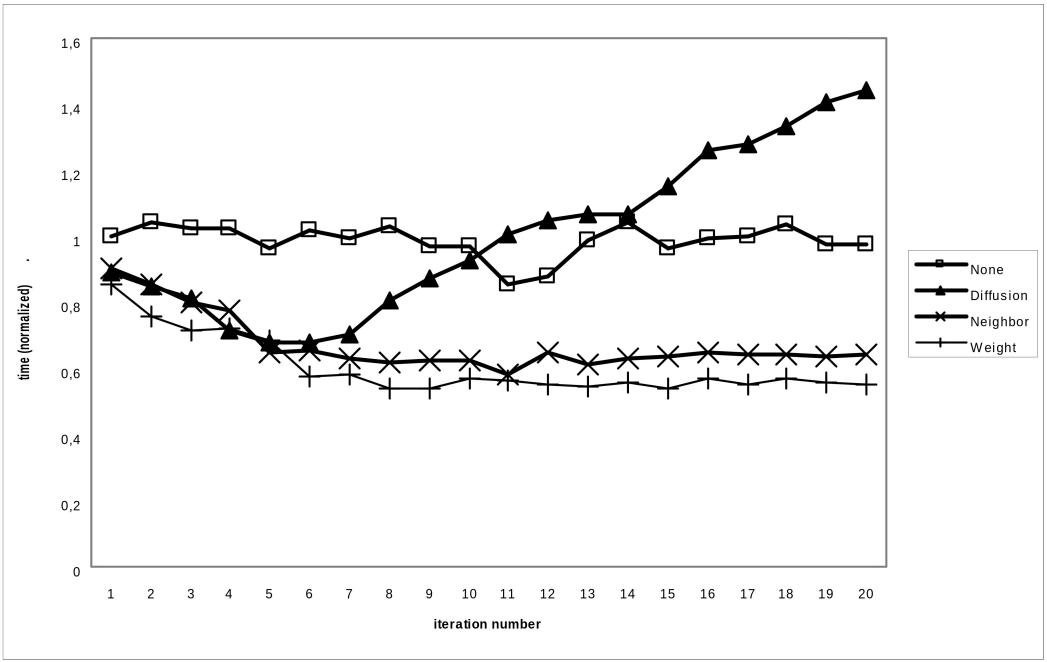


Fig. 11 The neighborhood relation influence on dynamic load balancing

In the graphics, four balancing versions are shown. The abscissa axis is the iteration number, the ordinate axis is the iteration execution time. The time is normalized, where 1 corresponds to the execution time of the first iteration of the *None* version.

The *None* version has no dynamic load balancing, the DFs not migrating. The execution time remains about the same, but a lot of the 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 not keeping the neighborhood relation, and the total execution time even exceeds the unbalanced version.

The *Neighbor* version is the diffusion load balancing with the neighborhood relation taken into account. Those DFs migrate, which have more neighbors in the target PE. Such a load balancing keeps the communication overhead on a certain level and is not growing with a lapse of time.

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

7. Conclusion

The LuNA system of fragmented programming is still under development and improvement. This approach provides a hardware undependable representation of numerical algorithms and their portability among a wide range of multicomputers. Therefore, our next step will be the development of a parallel numerical subroutine library on the basis of the algorithms fragmentation and the LuNA system for the LFP construction. We are also planning to support the GPU/FPGA extensions as execution units of computation nodes.

References

1. Valkovskii VA, Malyshkin VE (1988) Parallel Program Synthesis on the Basis of Computational Models. Nauka, Novosibirsk
2. Glushkov VM, Ignatiev MV, Myasnikov VA, Torgashev VA (1974) Recursive machines and computing technologies. IFIP Cong, Vol.1., pp. 65–70. North-Holland Publish. Co
3. Torgashev VA, Tsarev IV (2001) Programming facilities for organization of parallel computation in multicomputers of dynamic architecture. Programmirovanie, No.4, pp. 53–67.
4. Cell Superscalar, http://www.bsc.es/cellsuperscalar. Accessed 15 November 2010
5. Charm++, http://charm.cs.uiuc.edu. Accessed 15 November 2010
6. Shu W, Kale LV (1991) Chare Kernel – a Runtime Support System for Parallel Computations. J Parallel Distrib Comput, Vol. 11, Issue 3, pp. 198–211
7. 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. 544–552
8. 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. 207–216

9. Foster I, Kesselman C, Tuecke S (1998) Nexus: Runtime Support for Task-Parallel Programming Languages. Clust Comput, Issue 1(1), pp. 95–107
10. 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
11. 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. 139–170
12. Benson GD, Olsson RA (1997) A Portable Run-Time System for the SR Concurrent Programming Language. Workshop Run-Time Syst Parallel Program (RTSPP)
13. Malyshkin VE, Sorokin SB, Chauk KG (2009) Fragmentation of numerical algorithms for the Parallel Subroutine Library. Springer Verlag, LNCS, Vol. 5698, pp. 331–343
14. The Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) project http://icl.cs.utk.edu/plasma. Accessed 15 November 2010
15. 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. 755–765