AN APPROACH TO THE DEVELOPMENT OF PROGRAMMING SOFTWARE FOR DISTRIBUTED COMPUTING AND INFORMATION PROCESSING SYSTEMS

V. P. Kutepov, V. N. Malanin and N. A. Pankov
Moscow Power Engineering Institute (Technical Univerity), Krasnokazarmennaya 14, Moscow, Russia

Keywords: Flow-graph stream parallel programming language, distributed computing systems, data-flow programming, scheduling.

Abstract: The problem of software development for distributed and parallel computing systems is considered in the paper. Its original issue is in extending our projecting work on parallel programming languages and operational tools for their realization on large scale computing systems.

1 INTRODUCTION

The problem of integration of distributed computing and information processing resources in order to achieve reasonable productivity and efficiency of their usage in practice is very complicated as well as far from satisfactory solution. Parallel computing systems, distributed databases and information systems are well known samples of the nontrivial distributed systems; each of them has its own problems and solutions. In this paper we are going to discuss the main features of our approach to development of software for distributed computing and information processing systems (DCIPS) that are of fundamental value when we look forward in order to outline contours of the future global distributed information systems in particular GRID. Many of stated in the paper are resulted from our experience of development of parallel programming languages and operational tools for their implementation on computer systems [2,3].

2 MAIN FEATURES AND PROBLEMS OF DCIPS

In this paper we intentionally use notion of DCIPS instead of GRID to focus on specific technical features of such systems not going into details on their social and economical matters. These features are:

- Multi-component structure with geographically distributed components and strictly specified functions;
- Dynamic structure alteration issued by the necessity for reconfiguration in order to manage load or because of components' failure;
- Parallelism referred to multiple asynchronous interactions between components;
- Relentless evolution of hardware and software base.

The main problems of DCIPS management and software development are:

- Integration of heterogeneous computing and communication hardware: computers, communication channels, storages, etc.;
- Creation of new parallel programming languages, instruments and operational tools supporting high level program development;
- Effective resource management and processes (jobs) scheduling as well as high reliability and fast failure recovery.

The first problem is being addressed to standardization and virtualization on different levels of hardware and software stack of DCIPS. Great contribution was made by the Grid community in the development of OGSI and its famous implementation Globus Toolkit (Foster I. et al, 2001). Actually the main Grid-tailored activities are in the integration area.
The second problem is of many aspects and besides development of high level programming languages

supporting different types of parallelism of real tasks and processes it suggests necessity of various instruments for debugging, verification, evaluation of complexity parameters and criteria for program's execution efficiency in distributed environment (Kutepov V.P et al, 1996).

The third problem is the most far from satisfactory solution for DCIPS such as GRID - large heterogeneous computing and information systems. Even for homogeneous computing systems, in particular clusters, various heuristics and on a knife edge solutions are used in order to get reasonable effect in scheduling processes and managing workload (Kutepov V.P., 2007).

In the following we summarize how approaches which we apply to solve these last two problems for clusters could be extended to DCIPS.

3 LANGUAGES AND INSTRUMENTS FOR DCIPS PROGRAMMING

The main purpose of a programming language is to reduce conceptual gap that always exists between a problem domain, ingenuous description of a problem or task, program developed to solve the problem and a computer system used for program execution. It's well known that the nearer programming language to the computer system – the more effective programs can be developed.

Next requirements are the most important for language oriented to support high level program development for DCIPS:

- Component-like style of program development;
- Well-defined forms of structural representation of program making the process of program development easy expressible; these structural forms should simplify process of debugging and analyzing of program as well as mapping it to DCIPS structure;
- Ability to support various forms of parallelism inherent in operation of DCIPS itself and programs created for them;
- Wide and flexible mechanisms for defining complex data structures and scaling structure of a program with the change of complexity of the data it should be applied to;
- Portability suggesting possibility of program component development by using the most suitable conventional programming languages.

None of currently existing DCIPS programming models fully comply stated requirements: MPI implementations lack well-defined structural representation of the program as well as good support for DCIPS parallelism; remote procedure call (RPC) and remote methods invocation (RMI) models including CORBA are rather low-level for parallelism representation and become very tricky when program reaches some level of complexity providing very limited capabilities for debugging.

In (Kutepov V.P et al, 1996) we have considered generalized architecture of programming environment for large scale computing systems which can be also retranslated to DCIPS.

Below we give short description of Flow-graph Stream Parallel Programming Language (FSPPL) that was developed for supporting high level component-like parallel programming for parallel computing systems (Kotlyarov D.V. et al, 2005) and which can be easily embedded in object oriented media of programming and adopted for the needs of DCIPS programming.

4 FLOW-GRAPH STREAM PARALLEL PROGRAMMING LANGUAGE

Formally, flow-graph parallel program (FGPP) in FSPPL is represented as a pair <FS, I>, where FS is a flow-graph scheme and I – interpretation.

FS = $<\{M_i \mid i=1, 2,...n, n\in N\}, P_{IN}, P_{OUT}\ C>$, where $\{M_i\}$ is a set of modules, P_{IN} and P_{OUT} are sets of all input and output points of all modules respectively and C is a connection function: $P_{IN} \times P_{OUT} \rightarrow \{true, false\}$.

Every module M_i is defined as M_i = <Name, $\{G_{INi_{m_i}}\}$, $\{G_{OUTi_{k_i}}\}$>, where $m_i,k_i\in N$, Name is a module name, $\{G_{INi_{m_i}}\}$and $\{G_{OUTi_{k_i}}\}$ are sets of named input and output groups of the module respectively.

Input group $G_{IN\ i_s}$ (s=1,2..m_i) of module M_i can be described as a pair <Name, $P_{IN\ i_s}$>, where Name is a name of input group, $P_{IN\ i_s} \subseteq P_{IN}$ – ordered set of input points of this group. Output groups are defined similarly with accuracy to the respective set of output points.

Interpretation I of flow-graph parallel program is a quadtuple I=<T, $\Im$, C_1, C_2>, where T = $\{t_i \mid i=1,2,... n, n\in N\}$ – set of data types, $\Im$ - set of methods or subprograms (Kotlyarov D.V. et al, 2005) associated with input groups of modules; C_1: $P_{IN} \cup P_{OUT} \rightarrow T$ is a points typifying function and C_2: $\{P_{IN_{i_{m_i}}} \mid i = 1..n\} \rightarrow \Im$ is a function associating a

method in $\Im$ with every input group of every module. It is supposed that method associated with input group $G_{IN\ i_s}$ (s=1,2..m_i) of module M_i has ordered set of formal parameters which types match with types of relative points in $P_{IN\ i_s}$. If a set of input points $P_{IN\ i_s}$ of input group $G_{IN\ i_s}$ is empty then corresponding method should have no formal parameters.
The representation of a module is given in figure 1 and it shows inputs and outputs of the module that are divided by groups and typified. Connections between outputs and inputs of modules of FS should satisfy condition that connecting points should be of the same type.

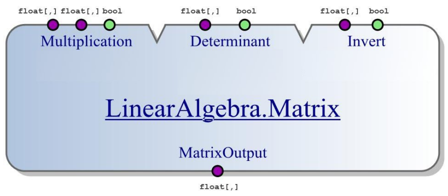

Figure 1: Visual representation of a module.

In figure 2 an example of flow-graph scheme is given which represents a program of assembly line for car manufacturing.

4.1 Operational Semantics of FGSPPL

Parallel execution process of FGPP is represented as a sequence of alternating states, where each state is characterized by a set of processes induced by invocation of methods from $\Im$ (Kotlyarov D.V. et al, 2005).

The process of FGPP execution is a consequence of the following rules:

- A FGPP module M_i is assumed to be ready for running by input group $P_{IN_{i_j}}$ with the data tuple, marked by tag t, if all its input points (in the corresponding memory buffers) have data marked by the same tag t.
- If module M_i is ready for execution by $P_{IN_{i_j}}$ with data tuples, marked by tags $t_1,\dots,t_k$, k processes, each of which is uniquely identified by the own tag t_j, can be simultaneously dispatched for execution.
- Modules with input groups without input points (which correspond to methods with an empty set of formal parameters) are assumed to be ready for execution by these input groups at the time of the FGPP execution initialization; however the processes induced by them can be generated only once.

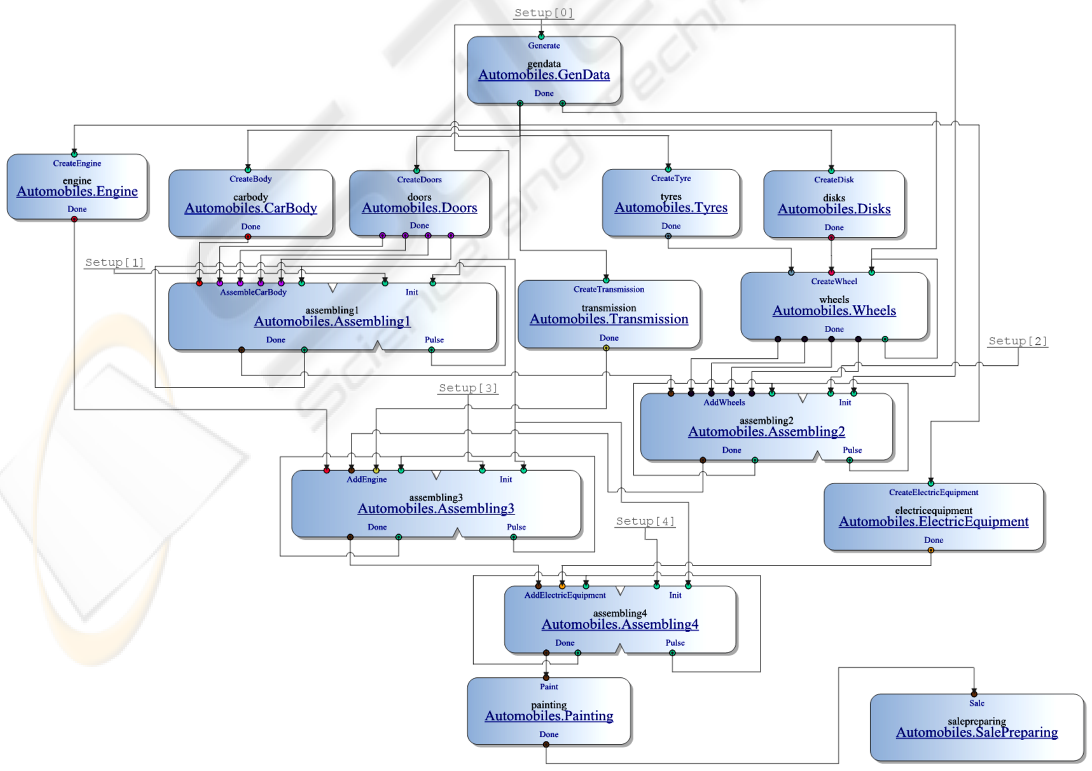

Figure 2: Flow-graph scheme for car assembly line.

- Process of FGPP execution is considered to be finished, when any module is not ready for execution and all the processes induced during FGPP execution are completed (terminated).
- During the execution method can use special-purpose functions *WRITE*, *READ*, and *CHECK*, which provide an interface between modules.
  a. *WRITE* function conducts write in the giving *output point* of the giving output group, with the specified *value* marked specified *tag*, and has the format: WRITE (<*name of output group*>, <*output*>, <*tag*>, <*value*>).
  b. *READ* function allows the process to read data with the indicated tag from buffers assigned to the input groups of the module that initiated the process. The data with indicated tag retrieved from the listed inputs are assigned to variables in the process - *input value*. Format of *READ* function: < *input value* > = READ (<*name of input group*>, <*input*>, <*tag*>). In the execution of *READ* function, if the requested data have not yet arrived to the buffer memory, the execution is delayed until the data arrive. The arrival time is controlled for any recording of data into the buffer memory of the corresponding input group. When *READ* function returns, the requested data are deleted from the buffer.
  c. For more sophisticated operation with data arriving to module input group (to be more precise, with the assigned buffers), the function <*availability flag*> = CHECK (<*name of input group*>, < *input*>, <*tag*>) is provided, which checks the availability of data with the indicated tags at the specified input point, and returns *TRUE* as result, if such data is present at the buffer, and *FALSE* – otherwise. This function allows the process to make a decision on its actions depending on the data availability.

The following forms of parallelism that objectively present in DCIPS and real-world tasks (Kutepov V.P et al, 1996) are easily represented in FGSPPL:

- Parallelism of data-independent fragments;
- Data parallelism (SIMD parallelism by Flinn) that is induced by simultaneous application of method to several differently tagged data tuples;
- Pipeline parallelism resulted from stream-like processing of data.

FGSPPL naturally combines the opportunities of gross-grain parallelism representation (that is realized at the level of the modules' methods) and fine-grain parallelism that is represented in methods of modules and can be implemented by multithreading.

Our experience in FGPPL programming shows that FGPPL allows to build adequate and often straight structure models of mass-service networks, distributed and many component systems, we have a positive experience in describing on FSPPL the distributed control processes of flexible automated manufactures, airports, etc, as well as multi-component program systems, where information relations are structured and permanent.

5 EMBEDDING FSPPL INTO OBJECT-ORIENTED ENVIRONMENT

Embedding FSPPL into object oriented environment is an important and apparent way to reach comprehensive requirements stated in part 3 of the paper, in particular to provide language support for complex data modelling. On the other side this embedding should enrich object-oriented language with dataflow superstructure on top for more flexible parallelism support. Such approach has already been considered as an important extension of object oriented systems (J.Paul Morrison., 1994).

Reusability of program code in object-oriented programming can be reached at different levels of abstraction starting from procedures, data structures, and classes and up to logically and physically connected sets of classes. Using these systems of classes in architecture of an application implies using of built-in object interacting schemas in runtime. These systems of classes delivering services in some area are usually called integrated libraries.

We've adopted the idea of integrated library to embed FSPPL into object-oriented environment. FGPP implemented with the library is called object-oriented FGPP (OOFGPP).

To build the integrated library for flow-graph stream parallel programming we have applied methods of object-oriented analysis for abstract FGPP, its syntax and parallel semantics. Using object-oriented decomposition we constructed the set of base classes that allows describing main objects of FGPP: scheme, module, input group, output group, input and output points and defined mechanisms of their interaction as private methods.

Now to build the OOFGPP developer has to implement a number of standardized abstract classes exposed by the library.

The embedding of FSPPL into OO environment enables developer to treat elements of parallel program as objects that allows using OO mechanisms such as polymorphism, encapsulation and inheritance while designing the parallel program. The largest impact this causes on the structure of data flows between modules: that is now the objects can be sent between modules as usual data. This implies that developer can send objects of different classes derived from one parent class from one module to another and depending on class different methods can be applied to this object inside module procedure due to polymorphism.

While designing the library we used the most general constructions that are available in any modern object-oriented platform. In current implementation we have chosen .NET CLR platform to build it.

For effective usage of this library it's important to deliver a full range of instruments that covers the whole process of OOFGPP design, implementation and execution on a target computer system (DCIPS). In the next part of the paper we will describe the developed environment for parallel programming based on the integrated library that includes all those instruments.

6 OBJECT-ORIENTED ENVIRONMENT FOR PARALLEL PROGRAMMING

The following requirements in implementation of the described above concept and language tools of object-oriented parallel programming were taken into account:

- Architecture of the environment should be built on component basis with strict division of the functions between them. Specifically, the following functions are strictly distinct in the developed environment:
  - Support of parallel program development process;
  - Remote access organization;
  - Management of parallel program execution process on DCIPS;
  - Management of processes and threads on DCIPS.
- The environment should use original algorithms for management of the workload, which allow supporting dynamically the effective usage of resources and decreasing parallel program execution time (see part 7 of this paper).
- The environment should be built as open and expandable, in particular the development of subsystem for fault tolerance ensuring parallel cluster work is carried out now as a new component of it.
- The software should have portability in its software realization and make availability to be applied on different computer platforms with different OS.

Let's briefly review the components of the developed environment.

1. Client software: FSPPL Integration Package.

This component is intended for managing interaction between user and the system. The component is integrated into popular integrated development environment (IDE) Microsoft® Visual Studio® (VS) 2005/8 and covers the full lifecycle of parallel program. It includes a VS project template that fully complies with the developed principles of design and realization of the parallel distributed programs using FSPPL. The component also contains specialized editor of the graph structures for creating and editing parallel program schemes. Program code on chosen programming language (VB, MC++, C#, J#) is generated based on the created schemes.

Module of program configuration provides services for setting the parameters of parallel program execution on cluster, or other DCIPS namely it allows developer to do initial mapping of the parallel program's scheme (its modules) to the nodes of the DCIPS (the analogue of machine file for MPI). Module of parallel program execution control data on load of the remote nodes to user and allows controlling the execution process of the parallel program.

2. Web interaction.

This component is a layer between of all distributed software components like client package and software for DCIPS nodes. The component is a Web-service performing two functions:

- Receiving user commands for the execution software and initiating appropriate actions;
- Providing to the client software an access to the data on nodes load and task completion status.

The component has access to the system database, which contains data on registered users and their tasks which were executed on the target DCIPS.

3. Program execution management software.

General principle of management architecture is a hierarchical decentralized organization when nodes of DCIPS are divided into groups (see figure 3).

Group server performs the following functions:

- Periodically obtaining data on load of the controlled nodes;
- Management of the group workload by relocation of the processes between nodes

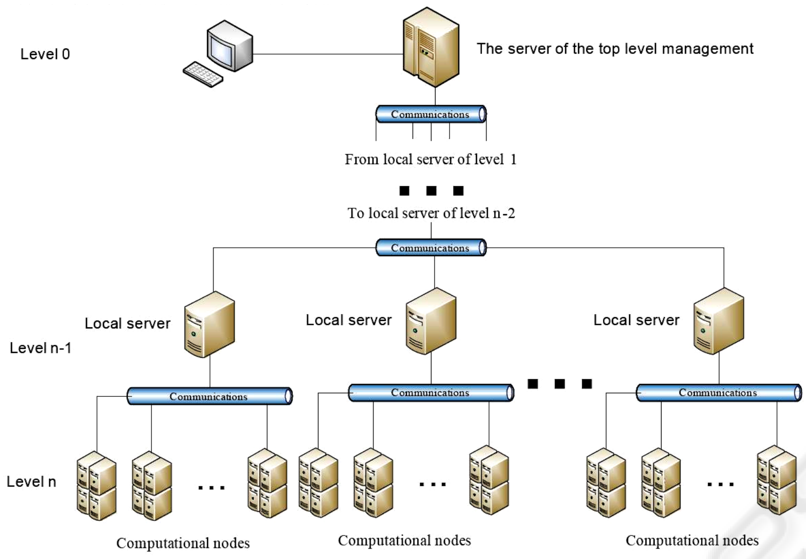

Figure 3: Architecture of DCIPS management.

based on obtained load data;

- Reconfiguration issued by failures or new systems discovery and administration.

Servers of higher levels perform the similar functions for subordinated groups.

Node management performs the following functions:

- Control of processes generation and, interaction with other processes, including those on other computers;
- Processes planning;
- Parameters measurement of the node load, processing of these parameters in order to forecast their future values and transfer to the server;
- Reaction to the server commands.

7 SCHEDULING PROCESSES IN COMPUTER OF DCIPS

Let us consider in what way the processes management at DCIPS nodes should be organized. The contemporary operating systems provide multi-tasking, using for this purpose the round-robin servicing discipline, which gives the advantage in the execution to short tasks or processes. It is the essential, if the user along with the programs execution carries out the debugging and other procedures and would like to receive the quick reply.

In figure 4 the scheme of processes service organization at node is shown, taking into account embedded scheduling processes in OS. In figure 4 the program block of the measurement of the workload parameters (WP) cooperating with the OS, implements the functions of the periodic measurement, averaging and forecasting of the computer workload parameters:

- Li(t) – workload of the i-th computer at the moment t, defined as the workload of its processor (the part time of its useful work);
- $\lambda i'(t)$ – intensity of the pages exchange with the disk memory;
- $\lambda i''(t)$ – intensity of inter-computer exchanges;
- $\lambda i'''(t)$ – intensity of the input-output commands occurrence in the running processes;
- Vi(t) – free memory of the computer;
- N(i)waiting(t) – set of the waiting for the execution processes.

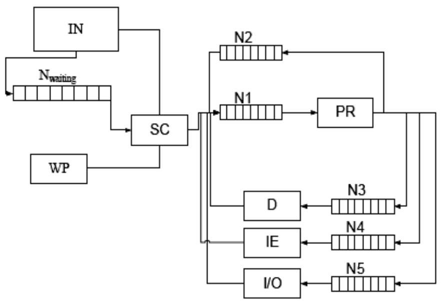

Figure 4: The organization of processes management at DCIPS node.

An interpreter (IN) of the parallel programs places the processes in the queue $N_{waiting}$ induced during execution of a program on processor (PR). A scheduler (SC) removes a part of these processes to the queue N_1 from which PR takes processes for execution in the round-robin order. The processes in the queues N_3, N_4, N_5 are waiting for execution of the exchange of pages with disk memory (D), the inter-computer data exchange (IE) and input/output (I/O) respectively. The SC can delay a part of being executed processes and place them in the queue N_2 in a case if a number of active processes $N_{active}=N_1 \cup N_2 \cup N_3 \cup N_4 \cup N_5$ is redundant and high swapping (great value of λ_i') is provoked.

The developed algorithm of scheduling processes on a node is given in figure 5. In this figure $\alpha_i = \lambda_i'/\mu_i$ - loading factor of the paging system (μ_i – paging system capacity). *A* – some experimentally derived threshold constant, by which the level of the computer workload is regulated.

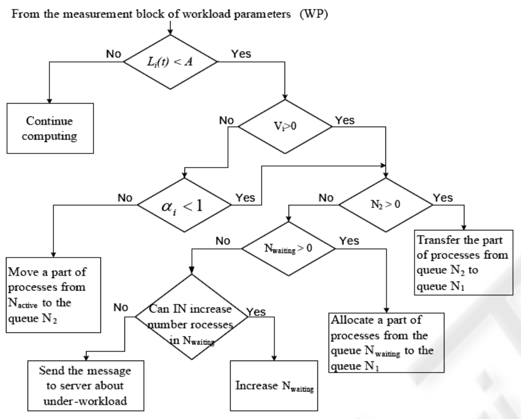
From the measurement block of workload parameters (WP)
No
$L_i(t) < A$
Yes
Continue computing
No
$V_i > 0$
Yes
No
$\alpha_i < 1$
Yes
No
$N_2 > 0$
Yes
Transfer the part of processes from queue N_2 to queue N_1
No
$N_{waiting} > 0$
Yes
Move a part of processes from N_{active} to the queue N_2
No
Can IN increase number rocesses in $N_{waiting}$
Yes
Allocate a part of processes from the queue $N_{waiting}$ to the queue N_1
Send the message to server about under-workload
Increase $N_{waiting}$

Figure 5: Scheduling processes in the computer.

8 SERVER FUNCTIONS IN MANAGING DCIPS WORKLOAD

The server is intended for the regulation of the nodes load, aiming to minimize the idle time of the processors due to the dynamic relocation of the processes and increasing the number of ready for execution on processes.

The server periodically obtains the data about the workload of its subordinated computers and forecasts its change. The scheme and the logic of the server interaction with the group computers are shown in figure 6.

In figure 6 the designations A and B on the arrows show the possible alternatives of the corresponding decisions about the redistribution of processes between computers during their dialogue with the server. At figure 6 all parameters of the load represent the averaged values on some interval and the forecast is the predicted values change of the same parameters.

The problem of the accurate measurement of the workload parameters is very important factor in scheduling processes and managing workload (Neil Gunther, 2005) in DCIPS. We performed wide the experimental investigation in order to better understand stochastic nature and the most significant parameters which characterize a behavior of the processes in computer systems (Kutepov V.P., 2007). As the result we developed the simple measurement and prediction workload parameters algorithms with small time consuming for their operation.

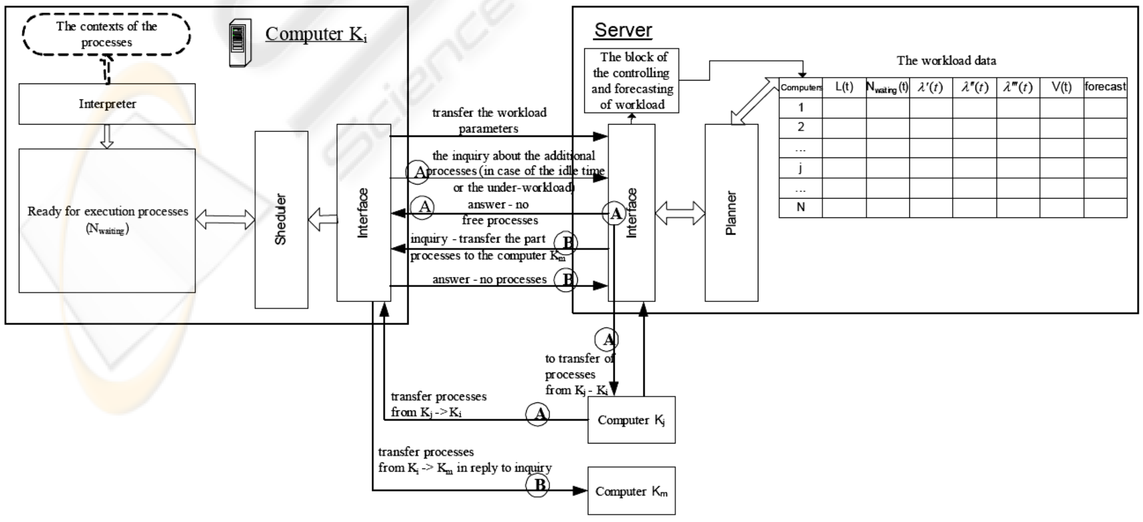
The contexts of the processes
Computer K_i
Interpreter
Ready for execution processes ($N_{waiting}$)
Sheduler
Interface
Server
The block of the controlling and forecasting of workload
The workload data
Computers
L(t)
$N_{waiting}(t)$
$\lambda'(t)$
$\lambda''(t)$
$\lambda'''(t)$
V(t)
forecast
1
2
...
j
...
N
transfer the workload parameters
the inquiry about the additional processes (in case of the idle time or the under-workload)
answer - no free processes
inquiry - transfer the part processes to the computer K_m
answer - no processes
Interface
Planner
A
B
to transfer of processes from K_j - K_i
transfer processes from K_j -> K_i
Computer K_j
transfer processes from K_i -> K_m in reply to inquiry
Computer K_m

Figure 6: Scheme of interaction between server and controlled nodes.

9 CONCLUSIONS

It is a great willing to conclude the paper with optimistic wave bypassing many stones on the road of the comprehensive solution of DCIPS programming problem. The dream of deep automation of programming process is immortal as well as difficultly reachable as we now acknowledge. Development of software for various DCIPS is more difficult task inheriting the main problems of previous one. Probably more intensive cooperation of the researchers' efforts is necessary in order to yield general conceptual and theoretical platform for DCIPS software development that allows to say that they performed work successfully. As for our project we are planning to integrate operational tools with mentioned above globus toolkit platform.

REFERENCES

Kutepov V. P., 1996 In. *«On Intelligent Computers and Large Scale New-Generation Computer Systems». Journal of Computer and Systems Sciences International, vol. 35, no. 5.*

Kotlyarov D. V., Kutepov V. P., Osipov M. A., 2005. In *"Flowgraph Stream Parallel Programming and Its Implementation on Cluster Systems", Journal of Computer and Systems Sciences International, Vol. 44, No. 1, pp. 70-89.*

Bazhanov S. E., Kutepov V. P., Shestakov D. A., 2005. In *«Functional Parallel Typified Language and Its Implementation on Clusters». Programming and Computer Software 31(5).*

Kutepov V. P., 2007, In *«Intelligent scheluding processes and controlling workload in computing systems». Journal of Computer and Systems Sciences International, vol. 46, no. 5.*

Dr. Neil Gunther, 2005. In *Load, Average Part I: How It Works. TeamQuest Corporation.*

J.Paul Morrison., 1994. In *Flow-based Programming: A new Approach to Application Development, Van Nostrand Reinhold.*

Foster I., Kesselman C. And Tuecke S., 2001. In *The anatomy of the grid: enabling scalable virtual organizations. International J. Of supercomputer Applications, N15 P.200-222, www.globus.org/research/papers/anatomy.pdf*