Code Generation for Multiprocessor Distributed Computing Systems

Victor Volkov, Vera Ivanova, Alexey Syschikov
Saint Petersburg State University of Aerospace Instrumentation
Saint Petersburg, Russia
{victor.volkov, vera.ivanova, alexey.syschikov}@guap.ru

Abstract—**Distributed computing systems is a well-known instrument to rise system performance and avoid bottlenecks of shared memory architectures. In some cases, distributed architectures are used in embedded systems: to simplify every single embedded processor, decrease power consumption, increase system tolerance and predictability, etc. However, when working with such systems, software developers face all kinds of complications related to distributed systems programming: workload distribution, communications implementation, computations synchronization etc. For embedded distributed systems it's impossibility to use distributed programming approaches like MPI: they are too "heavy" for embedded systems. We propose the extended application of VIPE visual programming technology that covers a software development for distributed multi-processor systems. VPL program structure and extensible code generation backend allows to automate most aspects of distributed programming and apply a single VPL programs both to shared memory and distributed systems without additional programming effort.**

I. Introduction

Distributed computing systems are complex multifunctional systems, which consist of several computing nodes (processors, modules or machines) connected with network communication channels. An effective programming of the multiprocessor distributed computing systems is a great challenge for developers. Such systems are oriented to solve a broad range of tasks: astronomy research [1], modeling of natural processes [2], solving resource-intensive tasks, telecommunications [3], mobile devices [4], networks [5] and so on. Many vendors such as NVIDIA [6] and Apple [7] participate in development of tools that are oriented to the distributed computing systems.

Distributed computing systems can be divided into centralized (a dispatching computing node is responsible for management of the system) and decentralized (the computing nodes are the autonomous modules interacting through a data exchange). Decentralized systems have recently gained a popularity due to the rapid growing of computer networks coverage and speed.

Our goal was to supply software developers for distributed systems with a programming technology that will allow the rapid development of distributed applications, ease the work with distributed computing configuration and management, and shield the developer from coding of networking interactions and synchronizations.

We base the work on the visual IDE VIPE to enable its programs to work on multiprocessor distributed platform that is combined into a single embedded system. To achieve this goal, we implemented an algorithm for task distribution over multiple processors, designed the mechanism of data exchange between processors and developed the set of rules for high-level code generation for each hardware platform.

II. VIPE IDE

The concept of VIPE (Visual Integrated Parallel Environment) [8] is based on the visual programming language VPL. VIPE is intended for the development of high-performance coarse-grained programs for parallel embedded systems. It allows creating domain-specific languages (DSL) for a particular domain at hand [9]. In addition, environment supports the OpenVX [10] framework, OpenCV and programming of microcontrollers of Atmel family [11].

The visual programming environment VIPE was developed to design programs (Fig. 1). It is the main interface of visual programming approach. IDE VIPE provides the technology and the tools for designing parallel algorithms, programming and debugging of multi-core embedded platforms, including heterogeneous ones.

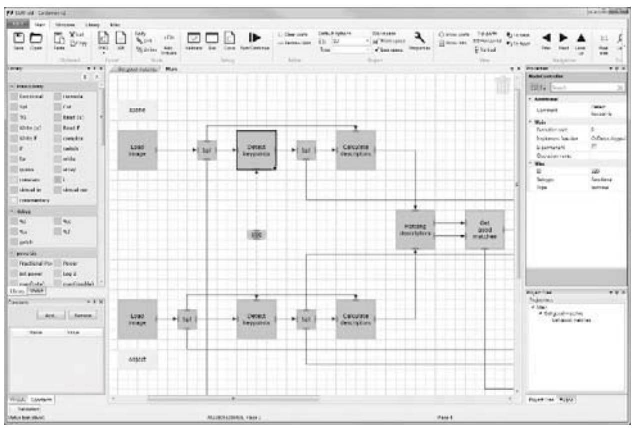


Fig. 1. VIPE IDE

A program scheme developed in VIPE IDE consists of various operators and links between them. The VPL language is used as the base language to create new domain-specific languages. VPL is based on the AGP model (Asynchronous Growing Processes) for dynamic parallel computations [12].

VPL is an “action language” [13] that can be executed as part of a model and translated into other languages. The AGP model defines the syntax of language, the semantics of language objects, and the design of control blocks. The formal model has a set of important capabilities such as formal verification, debugging and portability.

The structure of the code generated by base code generator for the visual programming language implies the existence of data processing operators, computation control operators, data-objects and data exchange links. Link is translated to a structure that stores intermediate data between operators during the program execution.

III. Code generation for distributed system

To execute a program developed in VIPE on a multiprocessor distributed computing system it is necessary to deal with several tasks of the allocating the program operators to processors, transferring data between them and generating functions of sending and receiving data.

VIPE contains a code generator module that is responsible for code generation. The main task of code generator is to translate VIPE program to a coarse-grained C-code, which will be executed on a specific hardware platform of a computing node of a distributed computing system.

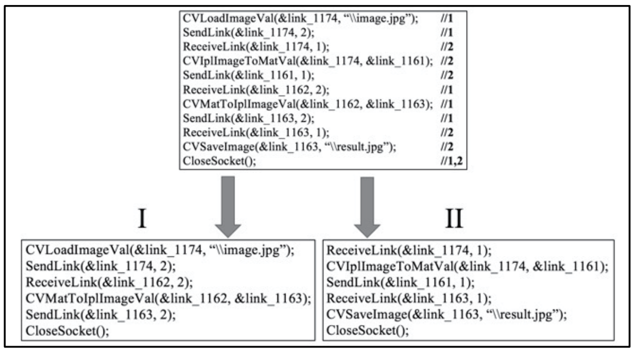
A. Allocation of program operators

First of all, an allocation for all operators on the processors of a distributed system should be defined.

We used cyclic Round-robin algorithm to allocate operators on processors. Round-robin algorithm was selected because it is simple and it can evenly distribute the workload among all processors. Also at the development stages it is useful to test the correctness of data transfers between all processors and correct generation of data exchange functions.

In practice, Round-robin algorithm is ineffective for distributed systems and in further releases this algorithm will be replaced with the set of adequate ones.

After that, the code will be distributed to separate files for each processor. These files will be separately compiled into the binary files for each computing device of a distributed system (Fig. 2).



```
CVLoadImageVal(&link_1174, “\\image.jpg”);        //1
SendLink(&link_1174, 2);                           //1
ReceiveLink(&link_1174, 1);                        //2
CVIplImageToMatVal(&link_1174, &link_1161);       //2
SendLink(&link_1161, 1);                           //2
ReceiveLink(&link_1162, 2);                        //1
CVMatToIplImageVal(&link_1162, &link_1163);       //1
SendLink(&link_1163, 2);                           //1
ReceiveLink(&link_1163, 1);                        //2
CVSaveImage(&link_1163, “\\result.jpg”);           //2
CloseSocket();                                     //1,2
```

```
CVLoadImageVal(&link_1174, “\\image.jpg”);
SendLink(&link_1174, 2);
ReceiveLink(&link_1162, 2);
CVMatToIplImageVal(&link_1162, &link_1163);
SendLink(&link_1163, 2);
CloseSocket();
```

```
ReceiveLink(&link_1174, 1);
CVIplImageToMatVal(&link_1174, &link_1161);
SendLink(&link_1161, 1);
ReceiveLink(&link_1163, 1);
CVSaveImage(&link_1163, “\\result.jpg”);
CloseSocket();
```

Fig. 2 Code generator logic for the distributed system

The code generator allocates the operators on the processors by assigning a processor's number for the code snippet with an operator call, which was previously defined by round robin algorithm. After that, each row of generated code will be put in a file that should be compiled into binary for each processor.

B. Data exchange between operators

Generated code is designed for decentralized multiprocessor distributed computing systems. Each node of the system is independent with direct interaction between nodes without using any centralized dispatching system.

Such peer-to-peer architecture is demonstrated on the Fig. 3. The presented links are “logical” and specify peer-to-peer data exchange: physically these links may be organized, for example, via one or more routers.

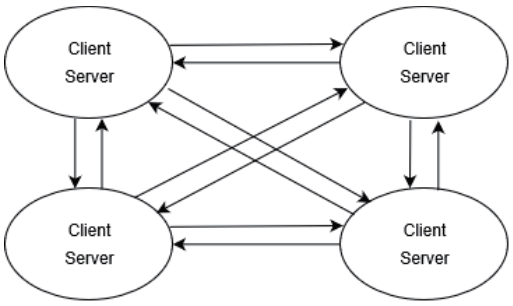


Fig. 3 Peer-to-peer architecture

It is necessary to transfer data correctly between operators that are located on different processors. The VPL program where each operator is marked with the processor’s number is shown on Fig. 4.

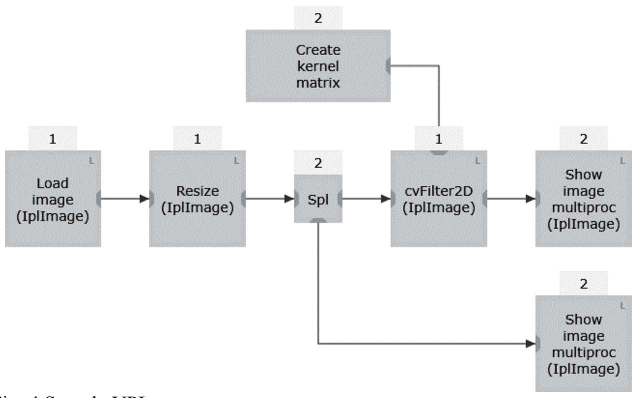


Fig. 4 Sample VPL program

If a pair of operators are connected with a link and are placed on different processors, it is necessary to organize data transfer between these processors.

Unlike the client-server architecture, each node in terms of data exchange must simultaneously perform the role of the server and the client.

There was developed a simple API for inter-processor communication (Fig. 5). It is a set of functions, which includes the initialization and the shutdown of a server; it also contains functions for sending and receiving data.

```
int InitSocket(int processor);
int SendLink(DataLink *link, int processor);
int ReceiveLink(DataLink *link, int processor);
int CloseSocket();
```

Fig. 5 Communication API functions

The communication API isolates code generator from software implementation of the network interfaces and drivers in a particular system and allows changing the functionality of data exchange depending on the specifics of the hardware and software platform. This adaptation can be done even by users of the platform without modifying the tools.

In the current implementation, data exchange is performed synchronously. The reason for this is based on the necessity of developing the prototype that will be applicable to any distributed platform, including ones with single-threaded execution and bare metal execution. Such platforms may not be able to provide asynchronous communications.

Socket interface using TCP protocol was chosen for a prototype implementation of communication between processes. It ensures correct data transferring and supported by most operating systems.

C. Integration of data exchange functions in code

Data transfer between processors is carried out by calling API functions. However, the logic of generation of sending/receiving functions should be clearly defined in order to provide correct execution of the distributed program. Consider a trivial fragment of the VPL program (Fig. 6) where two linked nodes are allocated on different processors, thus it is necessary to transfer data between the processors.

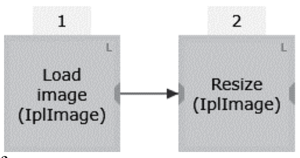


Fig. 6 VPL program fragment

Generated code for two processes is presented on Fig. 7.

```
I                                  II
CVLoadImageVal(&link_1, …);        ReceiveLink(&link_1, 1);
SendLink(&link_1, 2);              CVResizeVal(&link_841, &link_1246);
```

Fig. 7 Generated code for communication functions between operators

The *SendLink* function takes two arguments: the address of the link variable that stores the data produced after "Load" operator execution, and the number of target processor which means the destination of data transfer.

The same is for the *ReceiveLink* function: we need to pass the address of the link that will containing the input data for operator "Resize" and the number of processor from which we should receive data.

Target processor must know the number of the processor that sends the data, because there may be situations where several processors are trying to send data to the same processor. In such cases, absence of the number checking may lead to an incorrect data transfer.

After the processor 2 (Fig. 6) has received the data from the processor 1, operator "Resize" will be executed on processor 2.

The code generator developed for multiprocessor distributed computing systems allows generating code for any number of processors within the platform. Fig. 8 shows VPL program that applies blur filter on image with two format conversions. The program is mapped on a distributed system with three processors.

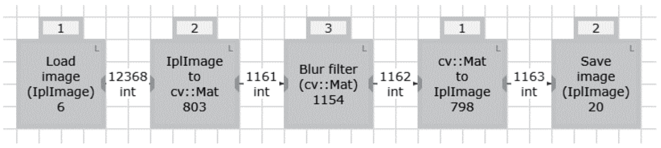
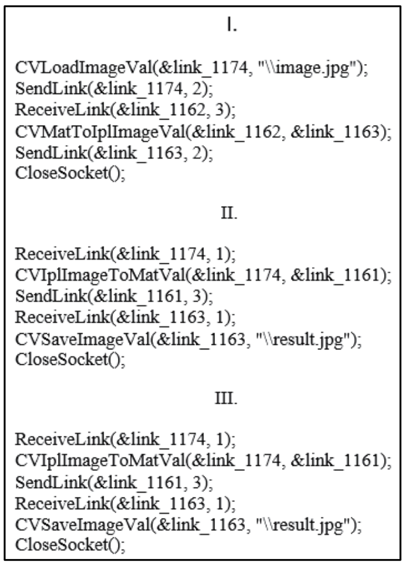


Fig. 8 Allocation of operators on three processors

The fragments of generated code (insignificant code fragments are removed) is presented on Fig. 9.



```
I.

CVLoadImageVal(&link_1174, "\\image.jpg");
SendLink(&link_1174, 2);
ReceiveLink(&link_1162, 3);
CVMatToIplImageVal(&link_1162, &link_1163);
SendLink(&link_1163, 2);
CloseSocket();

II.

ReceiveLink(&link_1174, 1);
CVIplImageToMatVal(&link_1174, &link_1161);
SendLink(&link_1161, 3);
ReceiveLink(&link_1163, 1);
CVSaveImageVal(&link_1163, "\\result.jpg");
CloseSocket();

III.

ReceiveLink(&link_1174, 1);
CVIplImageToMatVal(&link_1174, &link_1161);
SendLink(&link_1161, 3);
ReceiveLink(&link_1163, 1);
CVSaveImageVal(&link_1163, "\\result.jpg");
CloseSocket();
```

Fig. 9 A code generated for three processes

Logic of code generation for three or more processors is the same as for two processors. The code generator checks the link between the operators and generates a pair of exchange functions for sending and receiving side.

Code generator was tested and successfully generated code for a distributed computing system consisting of up to sixteen processors.

D. Locks and parallel execution

Presented programming technology for the distributed computing systems is designed for a wide variety of systems, including systems with a very limited performance capacity (embedded systems with a message passing). Such systems can function without an operating system (OS) or with a real-time operating system (RTOS) carrying a minimal set of services and lacking multitasking in order to spare resources. Such systems may lack a capacity to implement asynchronous data exchange. So a synchronous exchange has been chosen as a basic exchange mechanism, because it has the minimal set of requirements to the system software.

During the prototype development we had passed through application of several methods of exchange routines generation. We think it's reasonable to present them and they should be briefly presented in this paper to demonstrate revealed inconsistences.

Within the first approach, in the code generator sending and receiving functions were generated independently of each other.

Send functions were generated immediately after the data was produced on the source processor. Receive functions were generated right before the execution of the operator that needs these data on the target processor.

The example that was considered above (Fig. 8) is the simplest one due to its linearity. So the generation of sending and receiving functions independently of each other successfully coped with its task. However, if scheme has two or more entry points (Fig. 10) such generation of exchange functions may lead to the inability to execute the program.

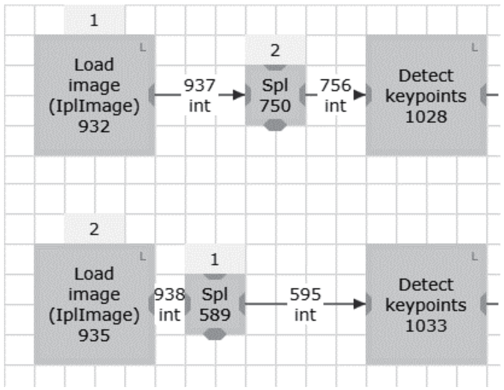


Fig. 10 Fragment of the scheme with two entry points

The code generator processes each scheme operator, checking the presence of input or output links. If there is an input link, receiving function must be generated before the calling of operator function. The processing of output links goes the same way: if there is an output link, connecting the operator with a node on another processer, sending function must be generated after the calling of operator function.

It can be seen that the processing of operators taking place in code generator occurs in the same order in which these operators are executed.

Let's consider the fragment of code generated for two processors (Fig. 11).

```
              I                                   II
CVLoadImage(&link_1, …);            CVLoadImage(&link_2, …);
SendLink(&link_2, 2);               SendLink(&link_1, 1);
ReceiveLink(&link_1, 2);            ReceiveLink(&link_2, 1);
```

Fig. 11 Result of an independent generation of exchange functions

After the operators 932 and 935 (Fig. 10) are executed, data exchange functions for adjacent operators are generated, because these operators are linked with operators that are placed on another processor.

Running such code will lead to the programs locking and inability to continue execution. Both processors will be trying to send data to each other. Further execution of both programs will be impossible because both processors will be waiting when another processor receive the data. It is the situation of a mutual locking (deadlock).

The second approach solved the deadlock problem with a synchronous generation of Send/Receive function pair. Now the rule of generation is as follows: send function must be generated after the execution of an operator and the correspondent Receive function should be placed into the current line of the code of the target processor.

Consider Fig. 10 again. Within the second approach, the generated code will look like this (Fig. 12):

```
              I                                   II
CVLoadImage(&link_1, …);            ReceiveLink(&link_1, 1);
SendLink(&link_1, 2);               CVLoadImage(&link_2, …);
ReceiveLink(&link_2, 2);            SendLink(&link_2, 1);
```

Fig. 12 A code generated after calling an operator

Program will finish its work correctly, however, it can be seen that an execution of operator with id=935 (Fig. 10) on a second processor will not start until data from first processor is received, even though this data does not affect the operation of this operator at all. So this approach may lead to a sequential execution of a distributed parallel program.

The last but not least approach allows avoiding locking issues and sequential execution of a program. A pair of exchange functions is generated before an execution of operator only if the operator has input links, connected with an operator on a different processor. Consider a code generated by these rules (Fig. 13):

```
              I                                   II
CVLoadImage(&link_1, …);            CVLoadImage(&link_1, …);
SendLink(&link_1, 2);               ReceiveLink(&link_1, 1);
ReceiveLink(&link_2, 2);            Splitter(&link_1, …);
Splitter(&link_1, …);               SendLink(&link_1, 2);
```

Fig. 13 A generated code for current realization

Both operators cvLoadImage on Fig. 13 do not generate a pair of exchange functions, because they have no input links. The following operators (Splitter) generate a pair of exchange functions, because there are input links, connecting operators on different processors. Thus, the nodes will be executed in parallel only when they are not limited by data dependencies.

It is worth mentioning that algorithm for generation of data exchange functions is not yet optimal and will be further refined. Besides, when using asynchronous exchange and dealing with its features, the methods of code generation can be modified both for optimization and to avoid potential problems.

E. Generation of complex programs

When VPL program has a complex structures and consists of hierarchically structured sections, it is necessary to pass the data inside these sections. Fig. 14 shows a program with one section:

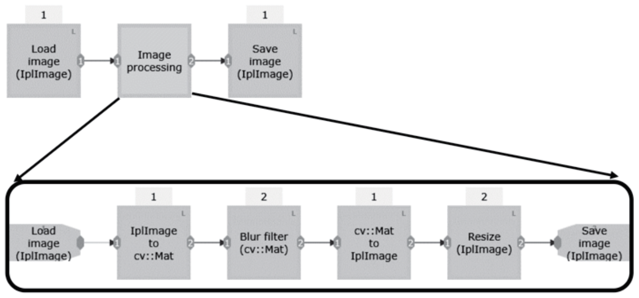
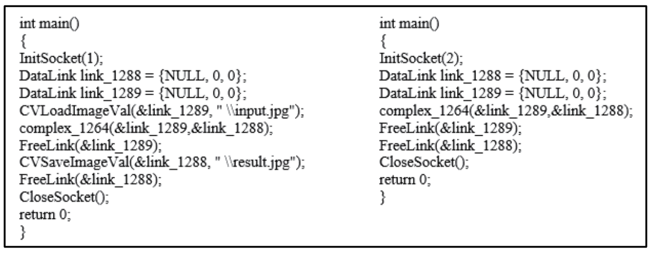


Fig. 14 VPL program with one section

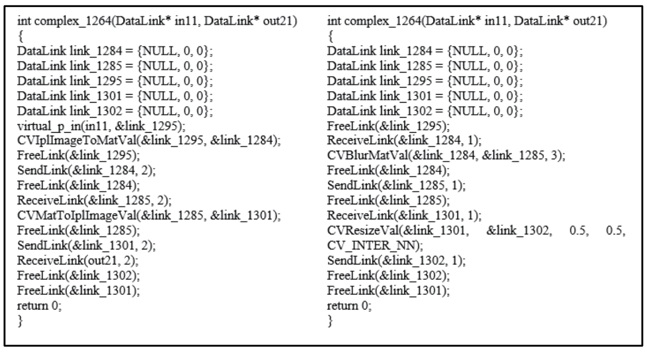
Fig. 15 shows a code generated for a main section and Fig. 16 shows a code generated for a nested section.



```
int main()
{
InitSocket(1);
DataLink link_1288 = {NULL, 0, 0};
DataLink link_1289 = {NULL, 0, 0};
CVLoadImageVal(&link_1289, " \\input.jpg");
complex_1264(&link_1289,&link_1288);
FreeLink(&link_1289);
CVSaveImageVal(&link_1288, " \\result.jpg");
FreeLink(&link_1288);
CloseSocket();
return 0;
}
```

```
int main()
{
InitSocket(2);
DataLink link_1288 = {NULL, 0, 0};
DataLink link_1289 = {NULL, 0, 0};
complex_1264(&link_1289,&link_1288);
FreeLink(&link_1289);
FreeLink(&link_1288);
CloseSocket();
return 0;
}
```

Fig. 15 Main section code for each processor



```
int complex_1264(DataLink* in11, DataLink* out21)
{
DataLink link_1284 = {NULL, 0, 0};
DataLink link_1285 = {NULL, 0, 0};
DataLink link_1295 = {NULL, 0, 0};
DataLink link_1301 = {NULL, 0, 0};
DataLink link_1302 = {NULL, 0, 0};
virtual_p_in(in11, &link_1295);
CVIplImageToMatVal(&link_1295, &link_1284);
FreeLink(&link_1295);
SendLink(&link_1284, 2);
FreeLink(&link_1284);
ReceiveLink(&link_1285, 2);
CVMatToIplImageVal(&link_1285, &link_1301);
FreeLink(&link_1285);
SendLink(&link_1301, 2);
ReceiveLink(out21, 2);
FreeLink(&link_1302);
FreeLink(&link_1301);
return 0;
}
```

```
int complex_1264(DataLink* in11, DataLink* out21)
{
DataLink link_1284 = {NULL, 0, 0};
DataLink link_1285 = {NULL, 0, 0};
DataLink link_1295 = {NULL, 0, 0};
DataLink link_1301 = {NULL, 0, 0};
DataLink link_1302 = {NULL, 0, 0};
FreeLink(&link_1295);
ReceiveLink(&link_1284, 1);
CVBlurMatVal(&link_1284, &link_1285, 3);
FreeLink(&link_1284);
SendLink(&link_1285, 1);
FreeLink(&link_1285);
ReceiveLink(&link_1301, 1);
CVResizeVal(&link_1301,    &link_1302,    0.5,    0.5,
CV_INTER_NN);
SendLink(&link_1302, 1);
FreeLink(&link_1302);
FreeLink(&link_1301);
return 0;
}
```

Fig. 16 Nested section code for each processor

The rules for code generation of a program with nested sections for multiprocessor distributed computing system are as follows:

- Section body and its function call are generated for all processors (even if there are no operators in the section allocated to this processor).
- Data exchange between processors is performed inside the section.
- When linked operators should be executed on the same processor, data exchange is performed via memory.
- When linked operators should be executed on different processors, data exchange is performed via generation of a pair of exchange functions.

Apart from code generation for simple programs, support for loops and conditional operators was added to code generator. Fig. 17 shows program scheme for computing the factorial.

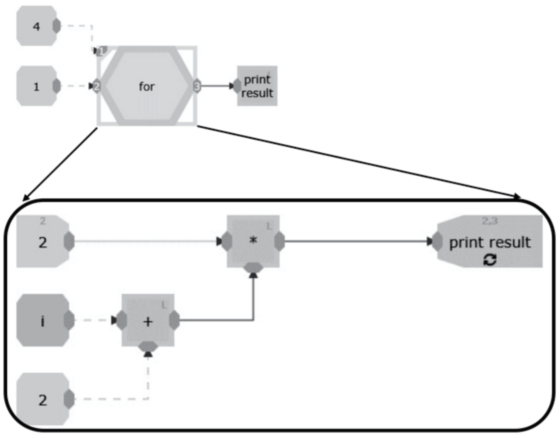
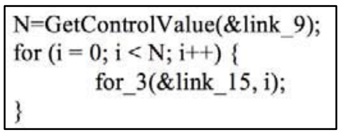


Fig. 17 VPL program for computing factorial

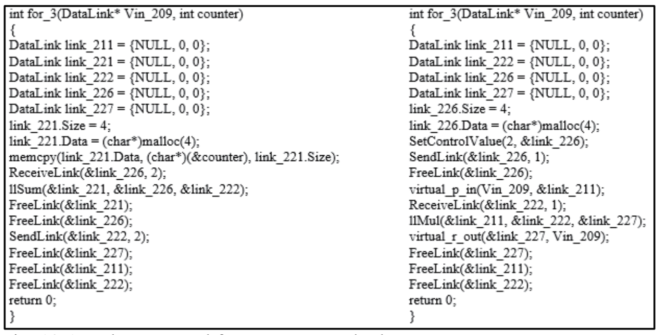
The call of the loop body function occurs on all processors. Number of iterations has to be passed to all processors. Next, there will be generated a loop construction in which the function of the For body is called. The following code will be generated for all processors (Fig. 18):



```
N=GetControlValue(&link_9);
for (i = 0; i < N; i++) {
        for_3(&link_15, i);
}
```

Fig. 18 A code generated for loop operator For

As the number of iterations has been broadcasted to all processors, all processors are already aware of it. All remaining generation rules are the same as for other nested sections. A code, generated for loop body is shown on Fig. 19:



```
int for_3(DataLink* Vin_209, int counter)
{
DataLink link_211 = {NULL, 0, 0};
DataLink link_221 = {NULL, 0, 0};
DataLink link_222 = {NULL, 0, 0};
DataLink link_226 = {NULL, 0, 0};
DataLink link_227 = {NULL, 0, 0};
link_221.Size = 4;
link_221.Data = (char*)malloc(4);
memcpy(link_221.Data, (char*)(&counter), link_221.Size);
ReceiveLink(&link_226, 2);
llSum(&link_221, &link_226, &link_222);
FreeLink(&link_221);
FreeLink(&link_226);
SendLink(&link_222, 2);
FreeLink(&link_227);
FreeLink(&link_211);
FreeLink(&link_222);
return 0;
}
```

```
int for_3(DataLink* Vin_209, int counter)
{
DataLink link_211 = {NULL, 0, 0};
DataLink link_222 = {NULL, 0, 0};
DataLink link_226 = {NULL, 0, 0};
DataLink link_227 = {NULL, 0, 0};
link_226.Size = 4;
link_226.Data = (char*)malloc(4);
SetControlValue(2, &link_226);
SendLink(&link_226, 1);
FreeLink(&link_226);
virtual_p_in(Vin_209, &link_211);
ReceiveLink(&link_222, 1);
llMul(&link_211, &link_222, &link_227);
virtual_r_out(&link_227, Vin_209);
FreeLink(&link_227);
FreeLink(&link_211);
FreeLink(&link_222);
return 0;
}
```

Fig. 19 A code generated for operator For body

The generation algorithm for loop operator While is mostly the same as for loop operator For.

The generation of conditional operators If and Switch also has a similar algorithm. Sample VPL program that contain If statement demonstrated on Fig. 20:

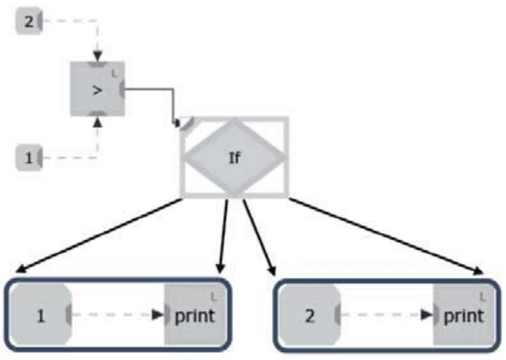
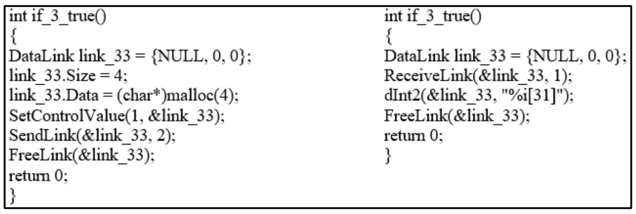


Fig. 20 VPL program with If operator

- Conditional operator's branches, executing when condition equals "true" and "false", are always generated for all processors (Fig. 21, Fig. 22).



```
int if_3_true()
{
DataLink link_33 = {NULL, 0, 0};
link_33.Size = 4;
link_33.Data = (char*)malloc(4);
SetControlValue(1, &link_33);
SendLink(&link_33, 2);
FreeLink(&link_33);
return 0;
}
```

```
int if_3_true()
{
DataLink link_33 = {NULL, 0, 0};
ReceiveLink(&link_33, 1);
dInt2(&link_33, "%i[31]");
FreeLink(&link_33);
return 0;
}
```

Fig. 21 Branch "true" of conditional operator If

```
int if_3_false()                          int if_3_false()
{                                         {
DataLink link_39 = {NULL, 0, 0};          DataLink link_39 = {NULL, 0, 0};
link_39.Size = 4;                         ReceiveLink(&link_39, 1);
link_39.Data = (char*)malloc(4);          dInt2(&link_39, "%i[36]");
SetControlValue(2, &link_39);             FreeLink(&link_39);
SendLink(&link_39, 2);                    return 0;
FreeLink(&link_39);                       }
return 0;
}
```

Fig. 22 Branch "false" of conditional operator If

- Condition is always broadcasted for all processors (Fig. 23).

```
SendLink(&link_23, 2);                          ReceiveLink(&link_23, 1)
conditionVar = GetControlValue(&link_23);       conditionVar = GetControlValue(&link_23);
```

Fig. 23 A code generated for condition broadcasting

- If statement is generated for all processors (Fig. 24).

```
if ((conditionVar)!=0)
if_3_true();
else
if_3_false();
```

Fig. 24 A generated code for If statement

Fig. 25 shows a code generated for main function of a program from Fig. 20:

```
int main()                                          int main()
{                                                   {
InitSocket(1);                                      InitSocket(2);
int conditionVar = 0;                               int conditionVar = 0;
DataLink link_21 = {NULL, 0, 0};                    DataLink link_21 = {NULL, 0, 0};
DataLink link_22 = {NULL, 0, 0};                    DataLink link_23 = {NULL, 0, 0};
DataLink link_23 = {NULL, 0, 0};                    link_21.Size = 4;
link_22.Size = 4;                                   link_21.Data = (char*)malloc(4);
link_22.Data = (char*)malloc(4);                    SetControlValue(2, &link_21);
SetControlValue(1, &link_22);                       SendLink(&link_21, 1);
ReceiveLink(&link_21, 2);                           FreeLink(&link_21);
llGreaterThen(&link_21, &link_22, &link_23);        ReceiveLink(&link_23, 1);
FreeLink(&link_21);                                 conditionVar = GetControlValue(&link_23);
FreeLink(&link_22);                                 FreeLink(&link_23);
SendLink(&link_23, 2);                              if ((conditionVar)!=0)
conditionVar = GetControlValue(&link_23);           if_3_true();
FreeLink(&link_23);                                 else
if ((conditionVar)!=0)                              if_3_false();
if_3_true();                                        CloseSocket();
else                                                return 0;
if_3_false();                                       }
CloseSocket();
return 0;
}
```

Fig. 25 A code generated for main section in scheme containing operator If

Switch operator is generated in a similar manner, apart from a greater number of conditional branches. Condition is also broadcasted for all processors, and functions for each condition body is generated and executed on all processors.

IV. CONCLUSION

The presented research of code generation approach and software prototype for a software development for distributed multiprocessor computing systems show promising results.

It brings the ability to apply once developed programs to both shared memory and distributed systems without program redesign. It shields developer from the duty to manually program interconnection routines, take care of workload distribution (in the mean of code writing), warn about synchronization and locks.

Further research and development will cover aspects of data interconnection optimization, application of existing VIPE analysis tools to analyze workload distribution efficiency, develop and allow to use more data interchange routines, including asynchronous functions and group interaction routines.

ACKNOWLEDGMENT

The research leading to these results has received funding from the Ministry of Education and Science of the Russian Federation within the project part of the federal assignment under contract N 8.4048.2017/4.6 from 31.05.2017.

REFERENCES

[1] Korpela, E., Werthimer, D., Anderson, D., Cobb, J., and Lebofsky, M. (2001). SETI@ HOME—massively distributed computing for SETI. Computing in science and engineering, 3(1), 78-83.

[2] V. V. Krzhizhanovskaya, V. V. Korkhov, M. A. Zatevakin and Y.E. Gorbachev "Parallel distributed computing in modeling of the nanomaterials production technologies." Parallel Computing Technologies (PAVT'2008): Proceedings of international scientific conference (St. Petersburg, 28 Jan±1 Feb 2008). Publ.: YUSU, Chelyabinsk. 2008.

[3] O. Huw, C. Edwards, and D. Hutchison. "The role of distributed computing in telecommunications: experiences and analyses." (1998): 106-110.i

[4] S. Balandin, M. Gillet, "Embedded Network in Mobile Devices", International Journal of Embedded and Real-Time Communication Systems (IJERTCS), vol. 1, № 1, 2010, pp 22-36

[5] Co-Modeling of Embedded Networks Using SystemC and SDL / V.Olenev, A.Rabin, A.Stepanov, I.Lavrovskaya, S. Balandin, M. Gillet // International Journal of Embedded and Real-Time Communication Systems (JERTCS) – Tampere. 2011. – #2(1) – C. 24-49

[6] NVidia official website, NVIDIA CUDA Technology Dramatically Advances The Pace Of Scientific Research, Web: http://www.nvidia.com/object/io_1229516081227.html

[7] Apple official documentation, Quick and Easy Distributed Processing, Web: https://documentation.apple.com/en/appleqmaster/distributedprocessingsetupguide/

[8] A. Syschikov, Y. Sheynin, B. Sedov, V. Ivanova. "Domain-Specific Programming Environment for Heterogeneous Multicore Embedded Systems". International Journal of Embedded and Real-Time Communication Systems (IJERTCS), vol. 5, . 4, 2014. pp. 1-23

[9] Ivanova, Vera, et al. "Domain-specific languages for embedded systems portable software development." Open Innovations Association (FRUCT16), 2014 16th Conference of. IEEE, 2014

[10] Syschikov, Alexey, et al. "Visual development environment for OpenVX". Open Innovations Association (FRUCT20), 2016 20th Conference of. IEEE, 2016

[11] Sedov, Boris, et al. "Domain-specific approach to software development for microcontrollers" Open Innovations Association (FRUCT17), 2015 17th Conference of. IEEE, 2015

[12] Ivanov, V., Y. Sheynin, and A. Syschikov. "Programming model for coarse-grained distributed heterogeneous architecture." XI International Symposium on Problems of Redundancy in Information and Control Systems: Proceedings, SUAI. 2007

[13] Y.E. Sheynin, A.Y. Syschikov, "Task-level Parallel Programming Language for Space and Aeronautical Applications", European conference for aerospace sciences (EUCASS), Moscow, 2005.