Joint NCC & IIS Bull., Comp. Science, 11 (1999), 1–14

A recursive parallel programming language and its application to algebraic computations*

Nikolay M. Badin German M. Brodskiy Valeriy A. Sokolov

This report presents an informal description of the GSTC language and some methods of recursive parallel programming language which enable us to organize parallelizing with the help of a set of basic structures (stencils). This makes it possible to design effective recursive parallel programs and to create, on their basis, some libraries of standard subprograms. The methods are intended, in particular, for organization of algebraic computations and are illustrated by recursive parallel programs for matrix multiplication.

1. Introduction

There are a number of approaches to the software development for sequential computer systems, a special place among them belongs to structured programming [1, 4, 5]. The development of software for parallel computer systems is much more complicated. The methods and means of programming proposed in this report are applied under the following limitations:

1) a recursive parallel form of a program representation;
2) dynamic parallelizing;
3) homogeneity of the computer system structure.

Both in sequential and parallel systems, the recursive method of programming is natural and convenient, it is very useful in designing program complexes from top to bottom.

However, to write a program in the recursive parallel style is not sufficient for its effective implementation. Such a parallel program should be written so that overhead expenses of organizing the recursive procedure calls, the generation and synchronization of parallel processes would be less than the amount of effective computations connected with parallel processes. The description of a recursive parallel C (RPC) language and a number of effective methods of recursive parallel programming are presented in [6]. Note that while the principles of recursive programming for sequential computer sys-

*This work is supported by the INTAS-RFBR (grant No 95-0378)

tems have been studied in detail [1, 3], the automation of recursive parallel programming raises a number of problems.

First, there are some difficulties in writing library procedures for multiprocessor systems with two-level memory. They are associated with different ways of data organization.

Second, in many cases modular programming for multiprocessor systems allows us to obtain only sequential-parallel programs [7] which are sometimes far from being the most effective. A simple example will make it clear. Let it be required to write a recursive parallel program for multiplication of three square matrices A, B and C of order N. It can be done by using two calls of the library procedure for multiplication of two matrices: for calculation of the matrix $D = AB$, and then $DC = ABC$. However, there is a more effective program that uses cutting of the matrix A into horizontal layers A_i of $K * N$ dimension with the subsequent calculation of every product A_iBC on the corresponding processor [2].

Third, in the case of recursive parallel procedures there are no methods similar to those of structured coding which permits one to obtain, from simple logic structures, programs convenient for testing, modifying and using.

Such research is actively being carried on at present. In particular, it is possible to mention Standard Template Library (STL) of the Rogue Wave company.

The article contains an informal description of some new methods for solving the above mentioned problems of recursive parallel programming and the GSTC (Generalized STencil C) language which supports these methods. Their capabilities are illustrated by an example of recursive parallel programs of matrix multiplication. Now a compiler for the GSTC language is under development, but its description is beyond the scope of the article.

The reader is assumed to be familiar with the basic operators of the C language.

2. Preliminaries

This section contains a description of some RPC language constructions [6], that will be further required.

A recursive parallel C language (RPC) is a subset of the standard C language, extended by special system calls (macrocommands). The expression "recursive parallel" derives from the corresponding programming style. The language satisfies the following requirements:

– it is a parallel programming language oriented to the architecture of a virtual multiprocessor system with dynamic parallelizing (a recursive parallel machine — RPM);

– a parallel program in the RPC language can be translated by the standard C compiler both to a parallel mode executed by RPM, and to a sequential code executed by a usual sequential computer;

– it can be used to study parallelism of programs (or their models), as well as the efficiency of RPM executing the programs (or their models).

The RPC language, as a subset of the standard C language, is a result of some restrictions. Many of them are due to the necessity of considering in the program the specific organization of a memory system which consists of local memory of the processors and the common shared memory.

The RPC language is based on parallel procedures described in a special way. The exchange of data with a procedure occurs through a particular block of parameters. All parallel processes generated by a parallel procedure should be synchronized at least by one operator Wait(). The parallel operators are the calls of a parallel procedure, the operators of access to the common shared memory, and some others that use the common system resources.

One of the RPC characteristics is a block of parameters containing local data for their transmission from a calling procedure to the called one, and vice versa. For the procedure parameter block with name *pname*, the corresponding data structure type named *Bpname* is declared. The name of a local variable of a given type (the name of a parameter block) is declared in the calling procedure. This name provides access to the elements of the parameter block. The access to the elements of the parameter block in a child procedure can be done only by the command P_(*elem*), where *elem* is the name of a structure element. An example of a recursive parallel program for summing array elements given below will make it clear. First, however, we present some more macrocommands of the RPC language.

The command of the title declaration for a parallel procedure is:

$$Parallel(pname);$$

where *pname* is the procedure name. It is used as the procedure title.

The command of a parallel procedure call is :

$$P\_call(pname, param);$$

where *pname* is the called procedure name; *param* is the parameter block name with the type *Bpname*. As a result of calling a parallel procedure by P_Call(), a potentially migratory process is generated which can be run at any processor.

The command for synchronization of parallel processes is:

$$Wait();$$

It suspends the execution of the procedure to wait for the completion of parallel processes started in the given procedure before executing $Wait()$.

Let us consider a recursive parallel program in RPC for calculating the sum $s = \sum_{i=0}^{n-1} a[i]$ of array elements $a[0], a[1], ..., a[n-1]$. A well-known algorithm is used: calculate the sum of the first $n/2$ elements (that is, the integer part of $n/2$) and, independently, the sum of the rest $n-n/2$ elements. The solution will be the sum of two partial results. Similarly we treat each of two "halves" of calculations, etc. The calculation of the sum of k array elements will be divided into two independent parts only if $k > vn$, where vn is the procedure parameter, by varying which it is possible to change the volume of calculations on "leaves" of the recursion tree and to choose experimentally the optimum value of vn. Now we can present the text of the program.

```
struct BSum                          /*The structure of a parameter block*/
  { int n,vn; float s,*a; }
Parallel(Sum); {       /*The declaration of the parallel procedure Sum*/
parameter blocks*/
                                              /*for recursive calls Sum*/
int i;
if(P_(n)>P_(vn)) {             /*The condition of recursive bootstrapping*/
  bp1.n=P_(n)/2;                         /*The preparation of parameters*/
  bp1.vn=P_(vn);                           /*of the first recursive    */
  bp1.a=P_(a);                             /*call                      */
  P_Call(Sum,bp1);                   /*The designation of the first process*/
  bp2.n=P_(n)-bp1.n;                     /*The preparation of parameters*/
  bp2.vn=P_(vn);                          /*of the second recursive    */
  bp2.a=bp1.a+bp1.n;                       /* call                     */
  P_Call(Sum,bp2);                  /*The designation of the second process*/
  Wait();                               /*Synchronization of the process*/
  P_(s)=bp1.s+bp2.s;           /*The "reverse" of the recursive execution*/
  }
else {
  P_(s)=0;                                 /*Computation on            */
  for(i = 0; i < n; i++)                   /*the "leaf" of the tree*/
    P_(s)=P_(s)+P_(a[i]);                  /*of the recursion          */
  }
} /* end of Parallel */
```

3. Skeletons, stencils and the language STC

In this section the STC (STencil C) language is described. It is a subset of the GSTC language considered in Section 4.

The STC language is the RPC language extended with some special constructions to describe a work layout for each specific procedure. The work layout for a particular procedure is agreed to be called the skeleton of the procedure. The STC language provides for the organization of the procedure skeleton as a sequence of stencils, that is, elementary modes of dividing work into some parts. A set of stencils recommended for use is included into the library of standard programs. Stencils can be parallel and sequential. The cause of using a sequential stencil, that is, an elementary mode of work layout not connected with parallelizing, can be, for instance, a limited volume of the local processor memory in a computer system with two-level memory organization. The sequential stencils are not considered in this report, so further the skeleton of a procedure can be understood as a way of recursive rolling, and the stencil, as the elementary way of recursive rolling.

To explain how the procedure skeleton, as a sequence of stencils, is organized, we consider the following example. Let it be required to write a recursive parallel program for multiplication of two square matrices A and B of order N. First, the matrix A can be slit into horizontal layers A_i of size $K*N$ by recursive rolling described by the first stencil. On the "leaf" of the recursion tree obtained for calculation of the product A_iB, it is possible, using the recursion again, to cut the matrix B into vertical layers B_j of size $N*L$, which is described by the second stencil. So, the procedure skeleton, being the sequence of two stencils, divides the calculation of the product AB into some parts, each of which is connected with the calculation of a product A_iB_j and is intended, in general, to work in its own processor module.

The STC language is a language of macrodefinition, and each stencil is a macrodefinition (macro) with parameters. Here the notion of a macro has more general meaning than is usually accepted. The STC language allows us:

1) to declare macros with parameters while some other macro can be used as a parameter;

2) to use the nesting of macros;

3) to declare a macro in the text both before and after its call.

Every STC command consists of three parts. It is obligatory for the first symbol to be $. It is followed by a key word defining the type of the command. Finally, depending on the key word, there are the concrete contents of the command. The commands will be considered below. Note that the symbol $ is used in the STC language, its extension GSTC, and some other cases. In particular, if $ stands before the macro name, it means that the macro's body is substituted for it here. As an example of a parallel stencil, consider the stencil *parvec*.

```
$stencil parvec(n,vn,Nameproc,bl1,bl2,bpi1,bpi2,Merger,Leaf)      /*1*/
if(P_(n)>P_(vn)) {                                                /*2*/
   bl1.n=P_(n)/2;                                                 /*3*/
   bl1.vn=P_(vn);                                                 /*4*/
```

```
    $bpi1;                                    /*5*/
    P_Call(Nameproc,bl1);                     /*6*/
    bl2.n=P_(n)-bl1.n;                        /*7*/
    bl2.vn=P_(vn);                            /*8*/
    $bpi2;                                    /*9*/
    P_Call(Nameproc,bl2);                     /*10*/
    Wait();                                   /*11*/
    $Merger;                                  /*12*/
    }                                         /*13*/
else { $Leaf }                                /*14*/
} /* end of stencil */                        /*15*/
```

Here the command $*stencil* of the STC language means the declaration of a stencil with the name *parvec* and parameters n, vn, etc. The parameter n corresponds to the dimension of the vector, with which some operations will be done (for example, summing its elements), vn is the maximum dimension of a subvector that cannot be further divided. The parameter *Nameproc* is the name of the called parallel procedure. Further there are the names of parameter blocks *bl*1 and *bl*2 which correspond to the first and the second recursive calls *Nameproc*. A number of parameters are declared here in the stencil (lines 3,4,7,8), the definitions of other parameters depend on a problem and are being set by a programmer with the macros *bpi*1 and *bpi*2. A macro *Merger* declares the "reverse execution" steps of the recursion, that is, merging of results obtained by parallel processes. The name *Leaf* belongs to a macro containing actions on a "leaf", when bootstrap of the recursion has been finished. The macros *Merger* and *Leaf*, as well as *bpi*1 and *bpi*2, should be defined by the programmer.

To declare a macro in the STC language, the command $*def* ... $*endd* is used, and to include a library stencil, the command $*ins*. More precisely, to call the first library stencil from a sequence of stencils forming the procedure skeleton, $*ins stencil*1 is used, to call the second stencil, $*ins stencil*2 is used and so on. Now we are able to write a program of summing the array elements not only in the RPC language, but in the STC language too.

```
struct BSum                      /*The structure of a parameter block*/
    { int n,vn; float s,*a; }
Parallel(Sum); {          /*The declaration of the parallel procedure Sum*/
struct BSum bp1,bp2;             /*The declaration of parameter blocks*/
                                         /*for recursive calls of Sum*/
int i;
$ins stencil parvec(n,vn,Sum,bp1,bp2,BPI1,BPI2,MERG,LEAF)
$def BPI1                        /*The definition of the macros BPI1*/
    bp1.a=P_(a);
$endd
```

```
$def BPI2                          /*The definition of the macros BPI2*/
   bp2.a=bp1.a+bp1.n;
$endd
$def MERG                          /*The definition of the macros MERG*/
   P_(s)=bp1.s+bp2.s;
$endd
$def LEAF                          /*The definition of the macros LEAF*/
   P_(s)=0;
   for(i = 0; i < n; i++)
      P_(s)=P_(s)+P_(a[i]);
$endd
}  /* end of Parallel */
```

4. Generalized procedures, the GSTC language and recursive parallel programs for multiplication of two matrices

The idea of using macrodefinitions received its further development in the GSTC language, an extension of the STC language. The GSTC language uses another type of macros with parameters — generalized procedures. A library of standard programs is formed of stencils and generalized procedures. A generalized procedure (g.p.) is not intended for direct calls from recursive parallel programs. They differ from typical procedures, above all, in multivalence, which is connected with the availability of so-called setting parameters. G.p.'s are intended for a preprocessor which can generate a procedure from the generalized one according to the values of setting parameters chosen by the programmer. Multivalence of the following three types can be involved in the g.p.

First, in the case of two–level memory organization, the g.p. can be multivalued as far as its memory (local or common) is concerned. Just as stencils fix the ways of the work layout, it is natural to have library access-to-memory methods. To do this in the GSTC language, a command

$$\$if \ < condition > \ ... \ [\$else] \ ... \ \$endif \qquad (1)$$

is used whose processing by the preprocessor makes it possible to eliminate some parts of the text and to retain others in accordance with the values of the setting parameters chosen by the programmer.

Second, the g.p. can be multivalued as far as the kinds of problems solved are concerned. For example, the use of command (1) and setting parameters provide a way of generating, from the same g.p., an effective procedure for multiplication of two matrices, an effective procedure for multiplication of a matrix by a column and an effective procedure for scalar multiplication of vectors.

Third, the g.p. can be multivalued with respect to the skeleton chosen by the programmer through specifying the values of corresponding setting parameters. This type of multivalence of the g.p. is of special interest. It will be considered in detail and illustrated by an example of the g.p. for multiplication of two matrices. Let us begin with the example. Let it be necessary to write the g.p. of calculating the product AB, where A is an $m * n$-matrix, and B is an $n * p$-matrix. If we want, with the help of the preprocessor, to generate from this g.p. an ordinary recursive parallel procedure for multiplication of two matrices and to apply it for m much less than t, and $p \geq t$ (t is the number of the processor modules of the computer system), we see that the first stencil of cutting the matrix A into horizontal layers is unsuitable. When generating an ordinary recursive parallel procedure from the g.p., the GSTC language allows us either to retain each stencil included into the skeleton of the g.p. in the generated procedure or to exclude it. This is organized by means of command (1) and the following technique: a setting parameter $st1$ is connected with $stencil1$ from the skeleton of the g.p., a setting parameter $st2$ is connected with $stencil2$ and so on. The value of each parameter is equal to 1, if the corresponding stencil is used, otherwise it is equal to 0. The name of an ordinary procedure generated by the preprocessor is formed from two parts (a prefix and a suffix). The prefix is the name of the g.p., and the suffix is a symbolic form for the decimal representation of the number $num1$ calculated by the preprocessor according to the formula:

$$num1 = st1 * 2^{k-1} + st2 * 2^{k-2} + \ \dots \ + stk$$

where k is the number of stencils forming the skeleton of the g.p. By choosing various subsequences of the sequence of stencils, a user can generate 2^k ordinary procedures from one g.p., whose names differ in the above mentioned suffix. For example, a sequential procedure without stencils will have the suffix 0. The possibility of calculating numbers (for example, the suffix $num1$) by the preprocessor is ensured in the GSTC language by the command $\$calc$. In order to tell the preprocessor that instead of the suffix $num1$ one should place its calculated value, the suffix is written as $\$num1$. To declare the name of the g.p. in the GSTC language, the command $\$gproc$ is used.

Passing to the design of a generalized procedure for multiplication of the $m * n$-matrix A by the $n * p$-matrix B, we first describe the algorithm to be used. The first stencil is intended for slitting the matrix A into horizontal layers A_i. The second stencil is planned to be used for cutting the matrix B into vertical layers B_J. Finally, to calculate A_iB_j, we use the third stencil intended for cutting matrices A_i and B_j, correspondingly, into blocks

$A_{i1}, A_{i2}, \ldots$ and $B_{j1}, B_{j2}, \ldots$:

$$A_i = \left(A_{i1} \mid A_{i2} \mid \ldots \right), \quad B_j = \begin{pmatrix} B_{j1} \\ B_{j2} \\ \vdots \end{pmatrix}$$

As is known, in this case $A_i B_j = \sum_l A_{il} B_{lj}$.

We present the text of the generalized procedure for multiplication of two matrices, where some obvious fragments are omitted. The matrices A, B, and $C = A * B$ are supposed to be arranged in the corresponding arrays a, b and c.

```
$gproc mulm(m,vm,n,vn,p,vp,a,b,c,st1,st2,st3)
{
$calc num3=st3;
$calc num2=num3+2*st2;
$calc num1=num2+4*st2;
struct Bmulm$num1                    /*The structure of a parameter block*/
{int $if(st1==1) vm, $endif
     $if(st2==1) vp, $endif
     $if(st3==1) vn, $endif
     m, n, p;
     float *a, *b, *c;
}
Parallel(mulm$num1) {
$if(st1==1)
    struct Bmulm$num1 bp11,bp12;/*The declaration of parameter blocks*/
                                /*for recursive calls mulm$num1        */
    struct Bmulm$num2 bp1;  /*The declaration of a parameter block for*/
      /*a call of the procedure mulm$num2 on a leaf of the recursion tree*/
   $ins stencil1 parvec(m,vm,mulm$num1,bp11,bp12,BPI11,BPI12,,LEAF1)
    $def BPI11
       . . .
    $endd
    $def BPI12
       . . .
    $endd
    $def LEAF1
       P_Call(mulm$num2,bp1);
    $endd
$else /* st1==0 */
$if(st2==1)
    struct Bmulm$num2 bp21,bp22;/*The declaration of parameter blocks*/
                                /*for recursive calls mulm$num2        */
```

```
    struct Bmulm$num3 bp2;  /*The declaration of a parameter block for*/
      /*a call of the procedure mulm$num3 on a leaf of the recursion tree*/
    $ins stencil2 parvec(p,vp,mulm$num2,bp21,bp22,BPI21,BPI22,,LEAF2)
    $def BPI21
        . . .
    $endd
    $def BPI22
        . . .
    $endd
    $def LEAF2
        P_Call(mulm$num3,bp2);
    $endd
$else /* st2==0 */
$if(st3==1)
    struct Bmulm1 bp31,bp32;        /*The declaration of parameter blocks*/
                                    /*for recursive calls mulm1          */
    struct Bmulm0 bp3; /*The declaration of a parameter block for a call*/
                /*of the procedure mulm0 on a leaf of the recursion tree*/
    int i;
   $ins stencil3 parvec(n,vn,mulm1,bp31,bp32,BPI31,BPI32,MERG,LEAF3)
    $def BPI31
        . . .
    $endd
    $def BPI32
        . . .
    $endd
    $def MERG
        for(i = 0; i < m * p; i++);
            P_(c[i])=bp31.c[i]+bp32.c[i];
    $endd
    $def LEAF3
        P_Call(mulm0,bp3);
    $endd
$else /* st3==0 */
. . . /*the body of a sequential procedure of multiplying m * n-matrix a*/
. . . /* by n * p-matrix b and writing the result into m * p-matrix c*/
$endif /* st3 */
$endif /* st2 */
$endif /* st1 */
}  /* end of gproc */
```

By using the preprocessor it is possible to generate, from the g.p., any of 8 different procedures for multiplication of two matrices: from the sequential procedure *mulm*0 to the procedure *mulm*7 that uses all of three stencils. As the text of the procedure *mulm*7 contains the call of the procedure *mulm*3 with a reduced skeleton and a block of parameters, the procedure *mulm*3, in turn, uses the call of a more simply organized procedure *mulm*1 and, finally, *mulm*1 calls the sequential procedure *mulm*0, it is advisable to generate, in addition to the procedure *mulm*7, the procedures *mulm*3, *mulm*1 and *mulm*0.

5. Sticking together the generalized procedures and a recursive parallel program for multiplication of three matrices

In the previous section we have already pointed out that the preprocessor makes it possible to generate a particular recursive parallel procedure from a generalized one. In this section a description of some other abilities of the preprocessor is given, which underlie the considered technology of recursive parallel programming.

Side by side with a command of generating a recursive parallel procedure from the generalized one, the CSTC language contains a command of specialization of the g.p. The use of the command assumes that the programmer assigns 0 to some setting parameters. Besides, the programmer can rename non-setting parameters of the g.p. A specialization command is processed in the following way. The stencils for which the corresponding setting parameters equal to 0 are removed, and the structure of the parameter block is processed in the same way. The text of the generalized procedure is transformed according to new symbols of parameters. The g.p. obtained is renamed by the programmer. The stencils remained in the g.p. are re-enumerated and the calculation of the number sequence *num*1, *num*2, ... is reorganized. Below we give an example showing in which cases the specialization of the g.p. is necessary.

A g.p. is called "prepared" if the names of its non-setting parameters are fixed. For instance, we have got the prepared g.p. as a result of execution of a command of the g.p. specialization by the preprocessor. Two prepared g.p.'s are called consistent if:

1) the structures of their parameter blocks are consistent, that is, the parameter types used in both blocks and the conditions under which they are included into the parameter block structure coincide (that is, the union of the parameter blocks is noncontradictory);

2) the skeletons of the g.p.'s contain the same number of stencils (say, k);

3) for every i $(1 \leq i \leq k)$, the stencils *stencil* i are of the same type

(say, *parvec*) and, for instance, for a stencil *parvec* the first two parameters must, respectively, coincide.

For consistent prepared g.p.'s there is a special command in the GSTC language such that the processor sticks the procedures together when processing. The consistency control of sticked prepared procedures is made by the programmer, as well as by the preprocessor. Without citing the bulky definition of the results of the sticking-together operation for two consistent prepared procedures, we only note that the sticking-together operation units the parameter blocks of sticked g.p.'s and changes the parameters on which this operation is done into working variables of a new prepared g.p. The skeleton of this new g.p. (named by the programmer) is constructed, in a definite way, from the skeletons of the sticked procedures. It contains as many stencils as there are skeletons of the sticked procedures, and the types of these stencils are the same. Finally, a "leaf" of the lowest level in the g.p. (that is, a fragment corresponding to a sequential program) is obtained by concatenation of the corresponding "leaves" of the sticked procedures.

Let us show now, using the operations of specialization and sticking together and the preprocessor, how to get, from the g.p. *mulm*, an effective g.p. for calculation of the product ABC, where A, B and C are matrices of size $m * n$, $n * p$ and $p * q$. This is done in three steps.

At the first step the specialization of the g.p. *mulm* is performed according to the given values $st2 = 0$ and $st3 = 0$ and to the redesignation of the parameter c to d. Having named the new g.p. *mumat* , we obtain a prepared g.p.

$$mumat(m, vm, n, p, a, b, d, st1). \tag{2}$$

At the second step we perform the specialization of the obtained g.p. *mumzt* which only renames some parameters: n becomes p, p becomes q, a becomes d, b becomes c and d becomes f. Having named the new g.p. *mumat*, we obtain a prepared g.p.

$$mumat(m, vm, p, q, d, c, f, st1). \tag{3}$$

At the third step the prepared generalized procedures (2) and (3) are sticked together over the parameter d. Having named the result of the operation *mtm*, we get the g.p. for multiplication of three matrices

$$mtm(m, vm, n, p, a, b, q, c, f, st1).$$

Note that the g.p. *mtm* is based on the effective recursive parallel algorithm for multiplication of three matrices mentioned in Section 1.

Besides the sticking-together command in the GSTC language, there are some other constructions which permit us to obtain some new effective g.p.'s from the available ones. For example, one of them makes it possible, with the help of the preprocessor, to obtain an effective g.p. for multiplication of r matrices from the g.p. for multiplication of two matrices.

6. Conclusion

The proposed technology of recursive parallel programming resembles structured programming. In particular, the design of every generalized procedure by means of a sequence of stencils is analogous to structured coding that permits us to construct arbitrary programs on the base of a limited set of the basic logic structures.

This technology gives us new tools of module programming for a multiprocessor system. We abandon ordinary requirements for program modules, in particular, the requirement of module independence from the context where it will be used. It is connected with the fact that by forming g.p.'s we ensure polyvariation of their usage. Unlike sequential programming, the result of processing such a generalized "module" becomes dependent on the source of input data and the usage of output data. Next, we have to abandon the formation of large programs without knowing the internal structure of the program module. We can console ourselves with the fact that not all information about module organization is needed but only the method of partitioning calculations, the skeleton. Finally, we abandon the attainment of independence between modules. It is reflected, for instance, in the sticking-together operation, a new kind of superposition of recursive parallel procedures.

The GSTC language underlying the new technology allows us to develop some g.p.'s which admit different versions of usage according to the type of work layout (in particular, the type of parallelizing), the type of data exchange and the type of the problems to be solved. The g.p.'s allow effective recursive parallel programs to be obtained in the case of a multi-staged recursion.

The practical value of the GSTC language and the suggested technology of the recursive parallel programming is defined by their feasibilities, namely

1) they point out the style of writing recursive parallel programs which increases their efficiency, reliability, and readability and ensures the independence from the programming product of its producer, which is necessary for design, support and modification of the programming product;

2) they allow a problem to be divided into some modules whose assemblage is made without a loss of efficiency, which gives us a possibility, in particular, of ensuring purposeful work of programmers;

3) they make it possible to develop and effectively use libraries of standard programs for multiprocessor systems;

4) they open the scope of the development of effective tools for automation of recursive parallel programming.

References

[1] S. Alagic, M.A. Arbib, *The Design of Well-structured and Correct Programs*, Springer-Verlad, 1978.

[2] N.M. Badin, G.M. Brodskiy, *An Approach to Recursive Parallel Programming*, Design, Modeling and Optimization of Complex Information Systems, Yaroslavl State University, 1993, 14–19 (in Russian)

[3] W.H. Burge, *Recursive Programming Techniques*, Addison-Wesley Publishing Company, 1975.

[4] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, *Structured Programming*, Academic Press, 1972.

[5] R.C. Linger, H.D. Mills, B.J. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Company, 1979.

[6] V.V. Vasil'chikov, V.P. Emelin, *Recursive Parallel Programming for Computer Systems with Dynamic Parallel Development*, Yaroslavl State University, 1992 (in Russian)

[7] Y. Wallach, *Alternating Sequential / Parallel Processing*, Springer-Verlag, 1982.